گزارش پروژه‌ی دوم درس هوش مصنوعی، شبکه‌ی عصبی

محمدآرمان سلیمانی، شماره‌ی دانشجویی: ۹۸۱۰۵۸۳۵

فهرست

مقدمه

در این گزارش به نحوه‌ی پیاده‌سازی پروژه‌ی دوم، شبکه‌های عصبی چندلایه، می‌پردازیم و نتایج به دست آمده را تحلیل می‌کنیم. برای پیاده‌سازی شبکه‌های عصبی از زبان پایتون و کتابخانه‌های Keras استفاده شده است.

همچنین برخی جاها از پیاده‌سازی پروژه‌ی اول درس کمک گرفته شده است، مانند ایجاد دیتاست نقاط برای قسمت‌های اول.

در این پروژه، ترتیب تحلیل‌های انجام شده در بخش‌های مختلف لزوما یکسان نیست و تفاوت‌هایی با شرح پروژه دارد، علت این امر این است که بنده مسیری که خودم طی کردم را در این تحلیل‌ها نوشته‌ام، مثلا ابتدا تعداد نورون‌ها را کم و زیاد کردم تا به یک مدل ثابت برسم و سپس تعداد نقاط ورودی را کم و زیاد کردم تا تفاوت‌ها روی مدل ثابت مشخص شود.

بخش اول: تابع یک بعدی بدون نویز

در پروژه‌ی اول یک فایل pointgen.py ایجاد شده بود که شامل یک تابع ریاضی بود و امکانات لازم برای ایجاد یک فایل csv حاوی نقاط تابع مد نظر. در این پروژه هم از همان فایل استفاده شده است. تنها نکته‌ی جدید این است که در این پروژه چون نیاز داریم تعداد رجوع زیادی به تمامی سطرهای دیتاست داشته باشیم و خیلی نیازی به کوئری زدن نداریم، به جای pandas از numpy استفاده می‌کنیم.

بررسی کد و یادگیری سه تابع با تنظیمات اولیه

```
9   min_x, max_x, point_cnt = 0, 5, 10000
10
11  # generate and load our function's points
12  generate_point_set(point_cnt, min_x, max_x)
13  dataset = loadtxt('points.csv', delimiter=',', skiprows=1)
14
15  # points are generated randomly, so a slice = random split
16  train = dataset[0:8000, :]
17  test = dataset[8000:10000, :]
```

در این قسمت بازه‌ی x نقاط و تعداد نقطه تایین شده و دیتاست ایجاد و بارگذاری می‌شود. x نقاط ما به صورت یک float با توزیع یکنواخت در بازه‌ی مد نظرمان انتخاب می‌شود و دیتاست ایجاد شده کاملا تصادفی است، بنابراین برای split کردن دیتاست نیازی نیست مجددا رندوم انتخاب کنیم و کافیست آن را به دو قسمت slice کنیم.

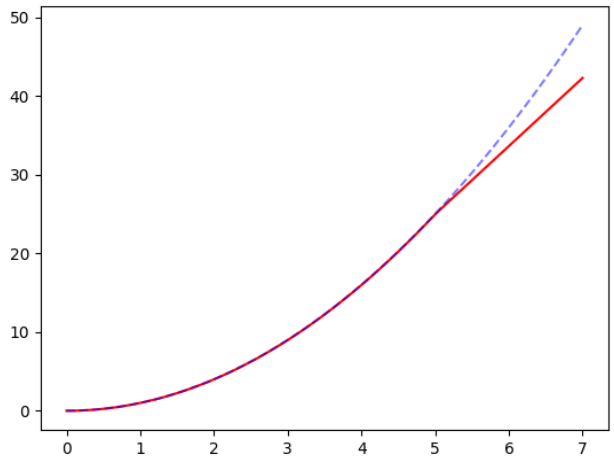
```
# generate NN
model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(100, activation='linear'))
model.add(Dense(50, activation='relu'))
model.add(Dense(1, activation='linear'))

model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_squared_error'])

model.fit(train[:, 0], train[:, 1], epochs=200, batch_size=200)

_, model_MSE = model.evaluate(test[:, 0], test[:, 1])
print('MSE: ' +str(model_MSE))
```

در این قسمت شبکه‌ی عصبی ساخته شده، train شده و تحلیل می‌شود. دقت داریم که در این قسمت رگرسیون انجام می‌دهیم و متریک ما MSE است و نه دقت. تنظیمات مربوط به شبکه‌ی عصبی در این بخش تغییر خواهند کرد و آنچه در این تصویر هست فقط یک مثال است.

```
# plot results + ground truth
plot_range = np.linspace(0, 7, 100)
predicted = model.predict(plot_range)
ground_truth = function(plot_range)
plt.plot(plot_range, predicted, '-r')
plt.plot(plot_range, ground_truth, '--b', alpha=0.5)
plt.show()
```

در نهایت تابع ایجاد شده (قرمز) به همراه تابع اصلی (خط‌چین) نمایش داده می‌شوند. بازه‌ی رسم توابع اندکی از بازه‌ی x دیتاست بزرگتر است تا توانایی مدل در برون‌یابی خارج از محدوده‌ی نقاط آموزشی نیز مشاهده شود.

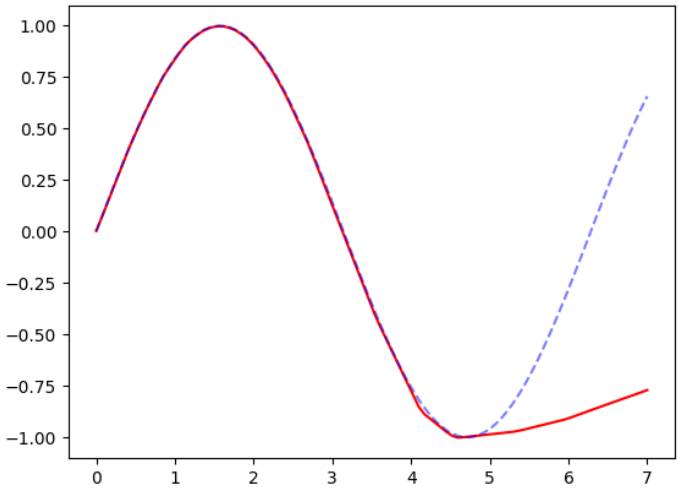
چنانچه با تنظیمات نشان داده‌شده سعی کنیم تابع x^2 را تخمین بزنیم خواهیم داشت:



و داریم MSE = 0.00032 (در بازه)

بنابراین خطای شبکه در بازه‌ی نقاط آموزشی (۰ تا ۵) خیلی کم است ولی در برون‌یابی فراتر از ۵ ناتوان است.

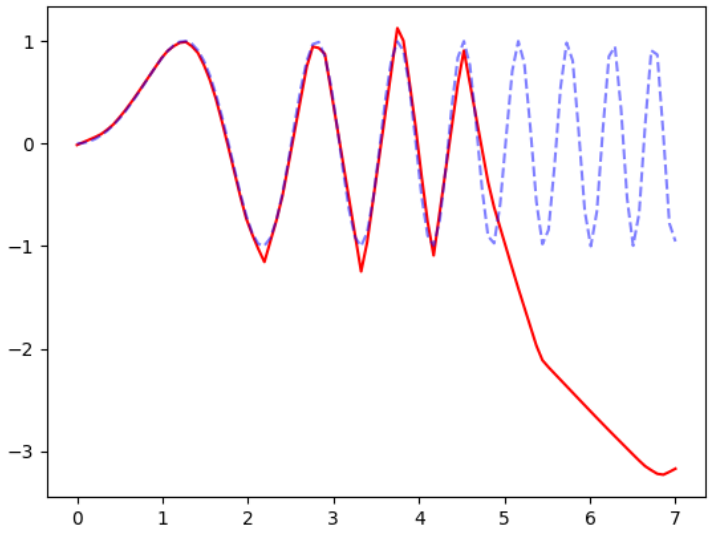
برای تابع sin(x) که تابعی پیچیده‌تر است با همین تنظیمات خواهیم داشت:



و داریم MSE = 0.00011 (در بازه)

بنابراین مجددا خطا در ناحیه‌ی آموزشی خیلی کم است ولی در برون‌یابی عملکرد خوبی نداریم.

حال تابع پیچیده‌تری انتخاب کرده و از sin(x^2) استفاده می‌کنیم. با همین تنظیمات داریم:



و داریم MSE = 0.01708 (در بازه)

اولا که برای این تابع هم خطا به میزان زیادتری بدتر شده، هم برون‌یابی همچنان بد است و هم اینکه مدل ما توانایی دریافت trend داده‌ها را ندارد (به وضوح مشاهده می‌شود که با زیاد شدن بسامد، منحنی تابع بدتر می‌شود).

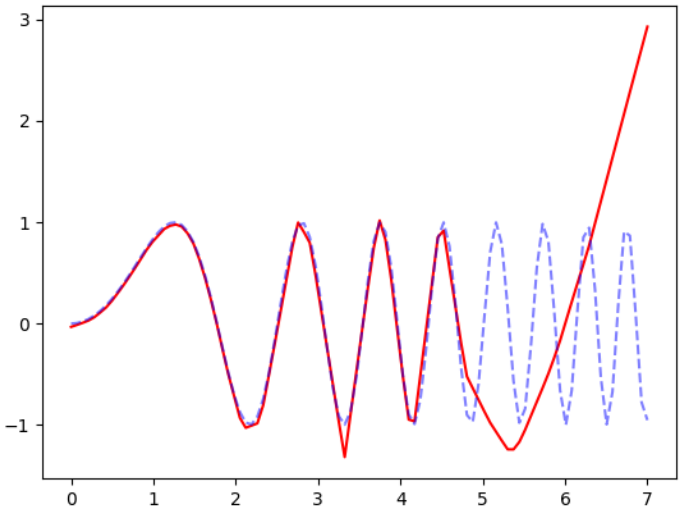
بنابراین به نظر می‌آید که باید شبکه پیچیده‌تر شود تا بتواند یادگیری بهتری داشته باشد.

تحلیل تعداد لایه‌های شبکه و نوع و تعداد نورون‌های هر لایه

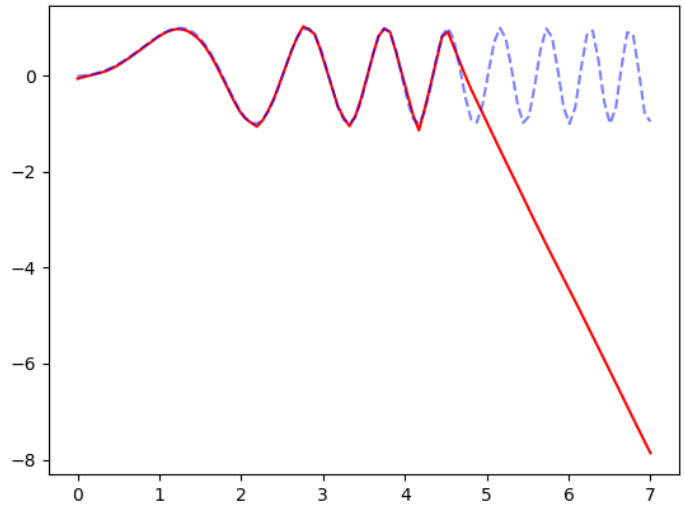
مشاهده شد که نسخه‌ی اولیه — ۴ لایه شامل ۲۰۰، ۱۰۰، ۵۰ و ۱ نورون — راهگشا نبود. از طرفی استفاده از نورون‌های ReLU در کارهای رگرسیون این‌چنینی خیلی درست نیست زیرا در یادگیری مقادیر و شیب‌های زیاد و غیرخطی دچار سختی می‌شود. بنابراین علاوه بر بررسی تعداد نورون و لایه‌ها، به نوع نورون‌ها نیز دقت می‌کنیم.

ابتدا شبکه را پنج‌لایه می‌کنیم و پیچیده‌ترین تابع را مجددا بررسی می‌کنیم:

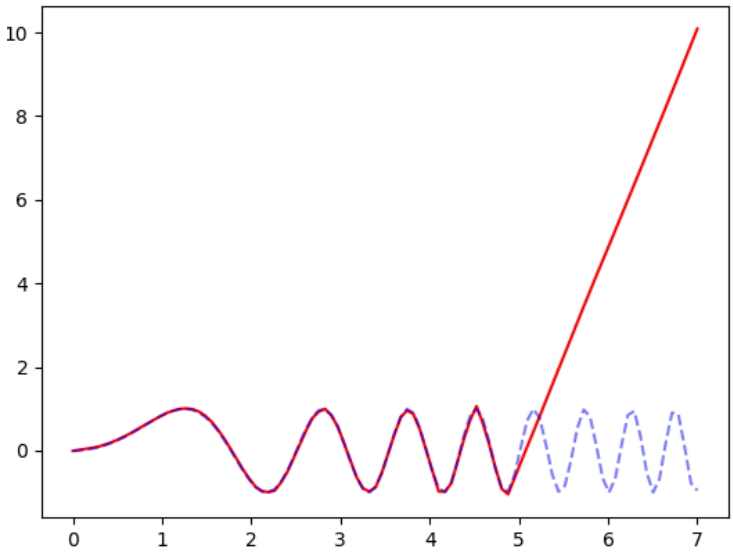
```python
# generate NN
model = Sequential()
model.add(Dense(200, input_dim=1, activation='relu'))
model.add(Dense(100, activation='linear'))
model.add(Dense(50, activation='relu'))
model.add(Dense(25, activation='linear'))
model.add(Dense(1, activation='linear'))
```



بنابراین هنوز هم مشکل برطرف نشده است. تعداد نورون‌ها در چهار لایه‌ی اول را دو برابر کرده و تکرار می‌کنیم:



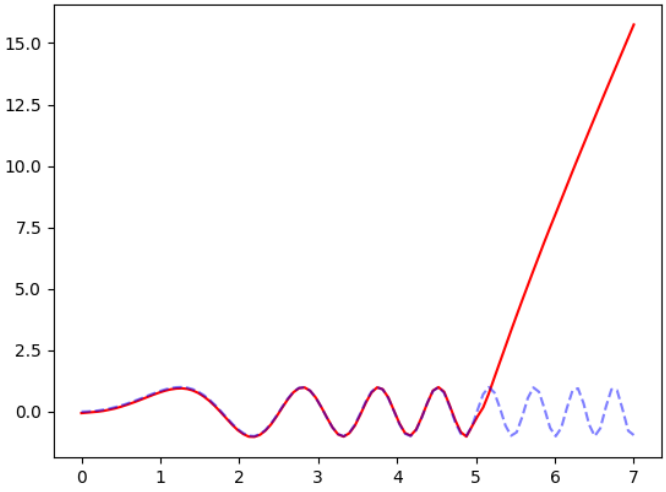
حال نورون‌های سه لایه‌ی اول را ۴۰۰ و چهارمی را ۱۰۰ قرار می‌دهیم. داریم:



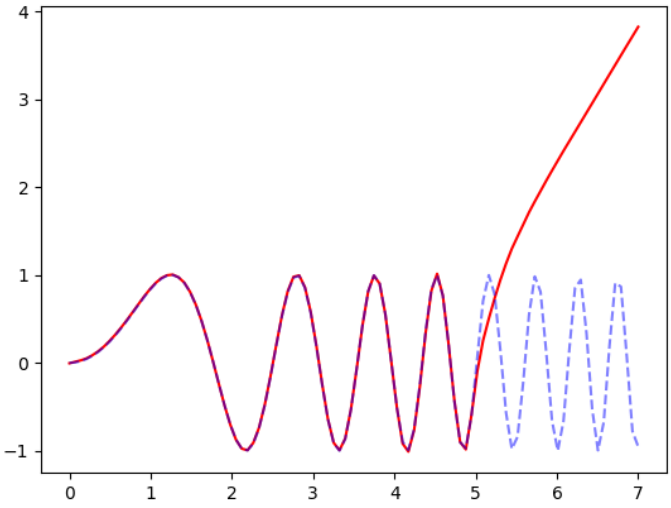
در این حالت MSE به 0.00142 می‌رسد که به مراتب بهتر از قبل است، و تابع لااقل در بازه‌ی آموزشی تمیز است.

بنابراین مشاهده می‌شود که اگر تعداد لایه / نورون کافی نباشد، مدل نمی‌تواند trend داده‌ها را یاد بگیرد. البته دقت داریم که تابع sin(x^2) سختگیرانه است زیرا بسامد با فرکانس متغیر دارد، و شبکه‌ای با ۵ لایه و جمعا ۱۵۰۱ نورون، شبکه‌ی خیلی کوچکی نیست. شبکه‌های خیلی کوچکتر می‌توانند دو تابع اول ما را به خوبی یاد بگیرند.

حال محض سرگرمی لایه‌ی چهارم را ReLU کرده و مجددا همین آزمایش را انجام می‌دهیم. داریم:



و MSE = 0.00161 که قدری زیادتر است ولی زیادتر بودن آن معنادار نیست. بنابراین خیلی هم بد نمی‌شود زیرا هنوز هم در لایه‌ی دوم و آخر، فعالساز خطی داریم. اگر لایه‌ی دوم هم ReLU شود داریم:

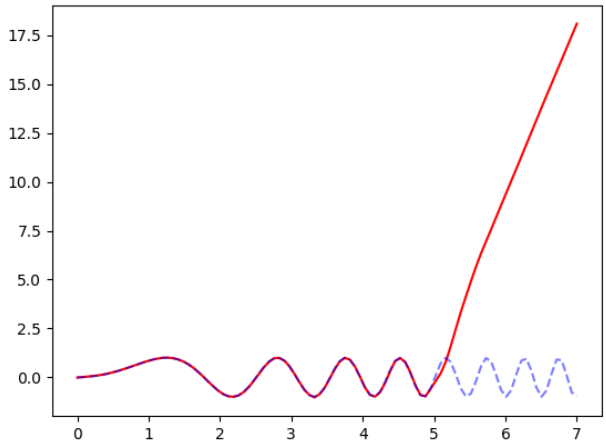


و MSE = 6.77e-5 که خیلی کمتر از قبلی‌هاست. بنابراین در این تابع استفاده از ReLU آسیب خاصی نزد و حتی به بهبود هم منجر شد که خلاف تصور بنده بود. انجام همین آزمایش روی x^2 هم نتایج خوبی داشت.

تعداد چرخه‌های شبکه برای تکمیل یادگیری

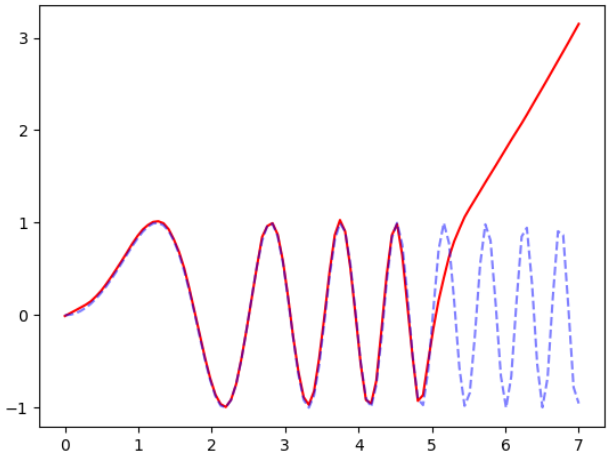
با توجه به MSE بسیار کم و یادگیری خوبی که در بازه‌ی نقاط آموزشی داشتیم، نمی‌توان انتظار داشت که با افزایش epoch یا اندازه‌ی batch یادگیری بهتر شود – با توجه به داده‌هایی که Keras هنگام یادگیری به ما می‌دهد، ما در اواسط یادگیری به چنین MSE می‌رسیم (یعنی همین الان هم در معرض overfit هستیم).

پس کاهش تعداد چرخه را امتحان می‌کنیم. در حال حاضر داریم:



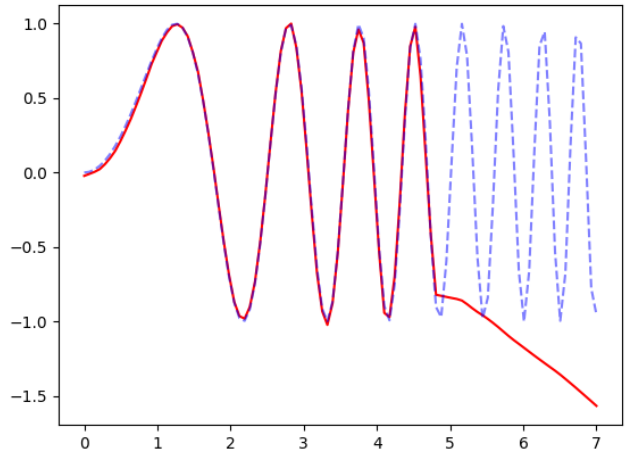
MSE = 0.00040

با کاهش تعداد epoch از ۲۰۰ به ۱۰۰ خواهیم داشت:



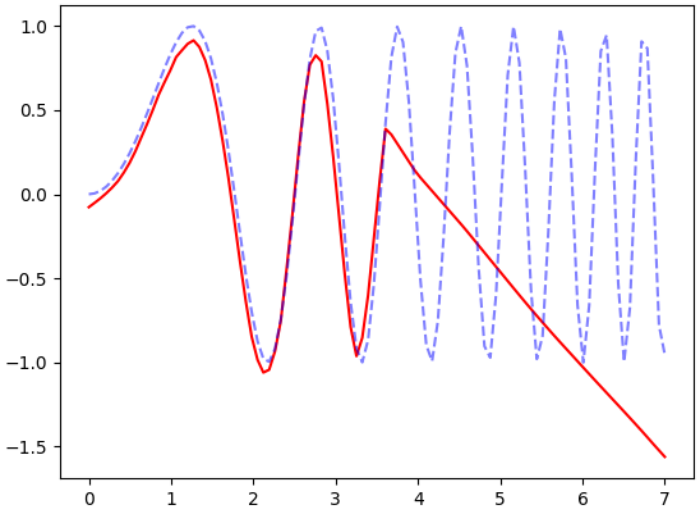
MSE = 0.00160

اگرچه افت کیفیت واضح است ولی هنوز خیلی هم بد نیست. با ۵۰ epoch خواهیم داشت:



MSE = 0.00309

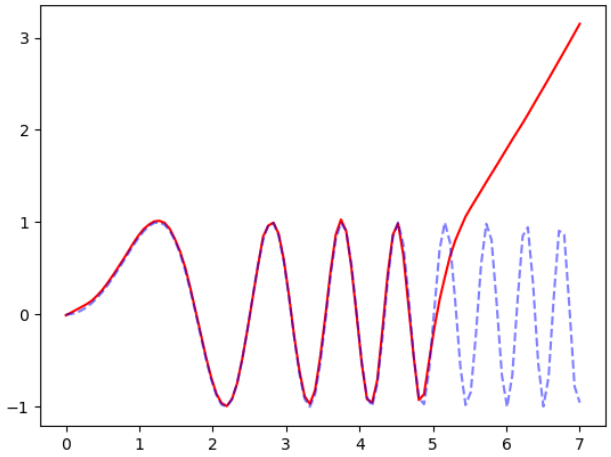
با کاهش تعداد epoch به ۲۰ خواهیم داشت:



MSE = 0.13002

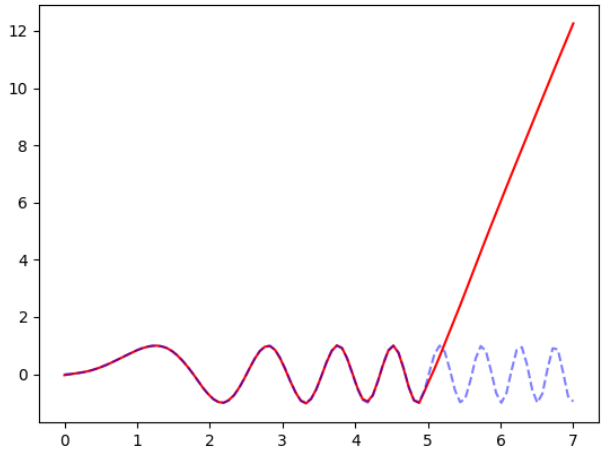
بنابراین به نظر می‌آید در اینجا حداقل ۵۰ دور برای یادگیری خوب لازم داریم و ۱۰۰ الی ۱۵۰ دور به بهترین نتیجه می‌رسد. حال با ۱۰۰ دور یادگیری، اندازه‌ی batch را تغییر می‌دهیم تا ببینیم چه تاثیری دارد.

با ۱۰۰ دور یادگیری و اندازه‌ی بچ ۲۰۰ داشتیم:



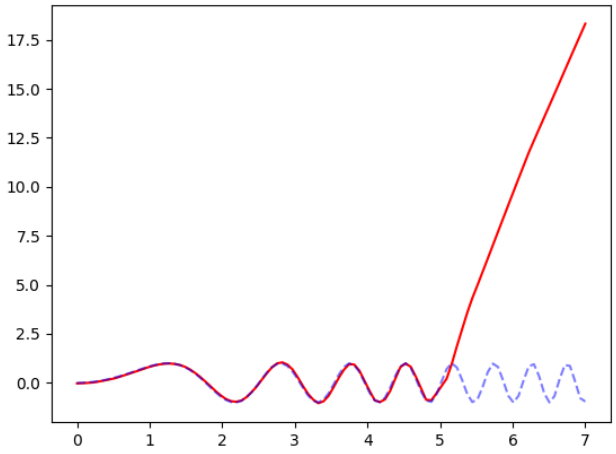
MSE = 0.00160

با کاهش از ۲۰۰ به ۱۰۰ خواهیم داشت:



MSE = 0.00035

می‌توان حدس زد که با ۱۰۰ و ۲۰۰ مدل overfit بوده که با ۱۰۰ و ۱۰۰ بهتر شده است. اندازه‌ی بچ را به ۵۰ کاهش می‌دهیم:

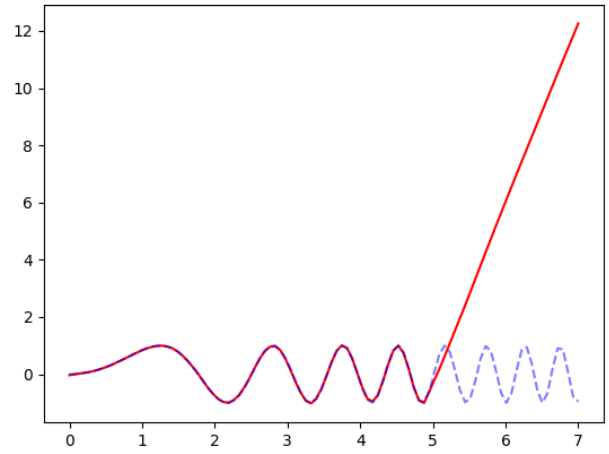


MSE = 0.0046

بنابراین افت مشاهده می‌شود. از ادامه دادن این روند صرف نظر می‌کنیم زیرا واضح است که مجددا کیفیت مدل کاهش خواهد داشت. بنابراین مدلی که تا اینجا انتخاب کرده‌ایم ۱۰۰ دور یادگیری و هر دور ۱۰۰ نقطه خواهد داشت.

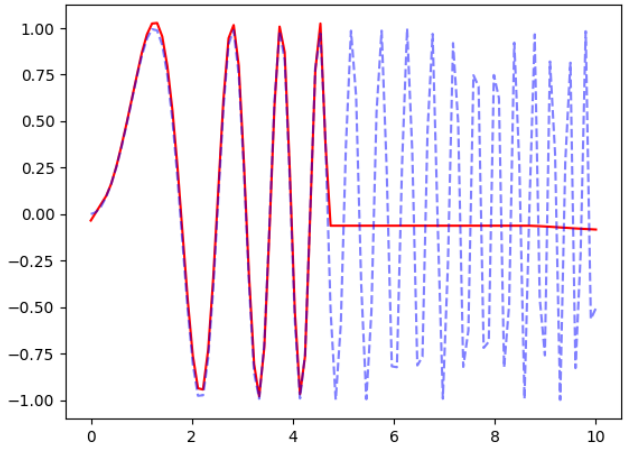
بازه‌ی نقاط ورودی

بازه‌ی کنونی ۰ تا ۵ است. برای این بازه داریم:



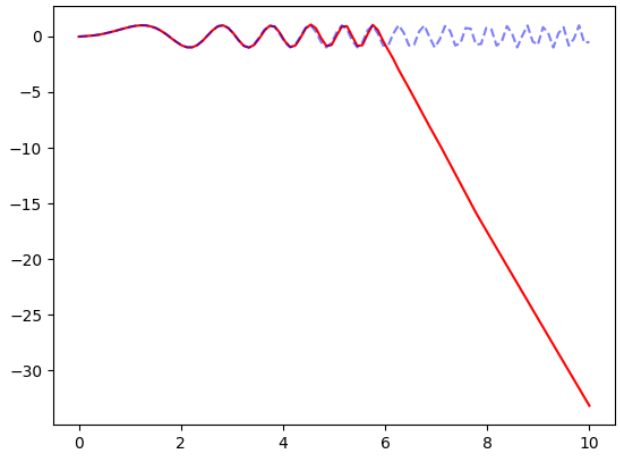
MSE = 0.00035

با ارتقای بازه به ۰ تا ۷.۵ داریم:



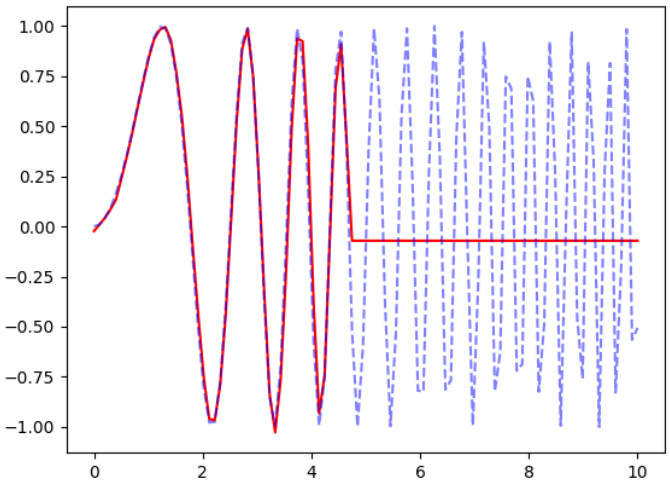
MSE = 0.19204

بنابراین یادگیری نقاط این بازه برای مدل ما مشکل است. اگر بازه را به نقاط ۰ تا ۶ کاهش دهیم خواهیم داشت:



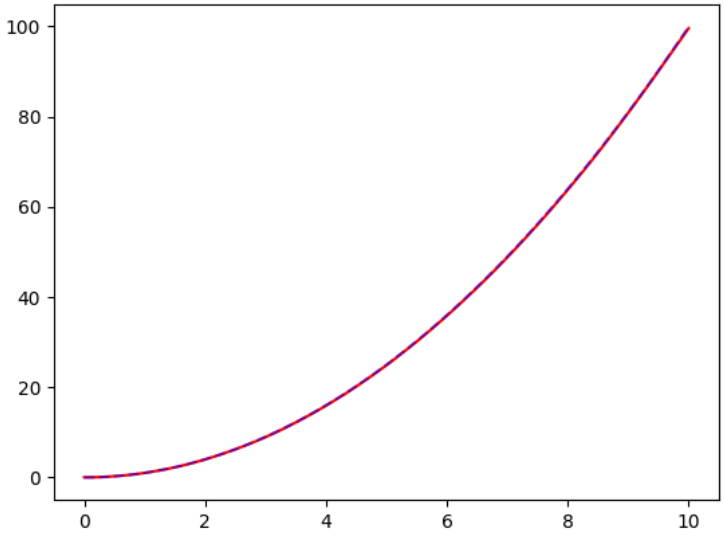
MSE = 0.02162

که خیلی بد نیست ولی از ۰ تا ۵ به وضوح بدتر است. اگر محض یادگیری بازه‌ی ۰ تا ۱۰ را بررسی کنیم:



MSE = 0.26211

بنابراین همانطور که واضح است، وسعت دامنه‌ی ورودی تاثیر به‌سزایی در یادگیری توابع پیچیده‌تر دارد. از آنجایی که توابع ساده‌تر یک روند ثابتی دارند، از گستردگی دامنه آسیب کمتری می‌بینند (مثلا اگر هدفمان یادگیری یک خط بود، اگر مدل در بازه‌ی ۰ تا ۵ موفق عمل می‌کرد، عملا در تمامی بازه‌ها موفق بود). به عنوان مثال اگر تابعمان x^2 بود:



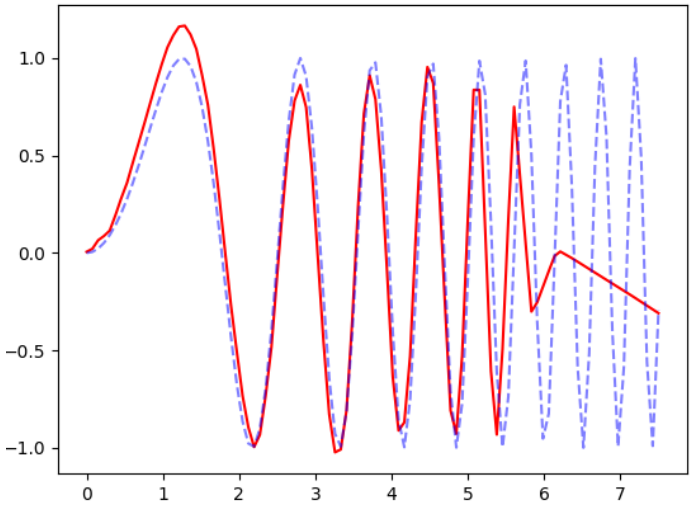
MSE = 0.01847

بنابراین افزایش دامنه وقتی آسیب جدی می‌زند که تابعمان در قسمت اضافه شده به دامنه، ویژگی جدیدی از خود نشان دهد.

تعداد نقاط ورودی

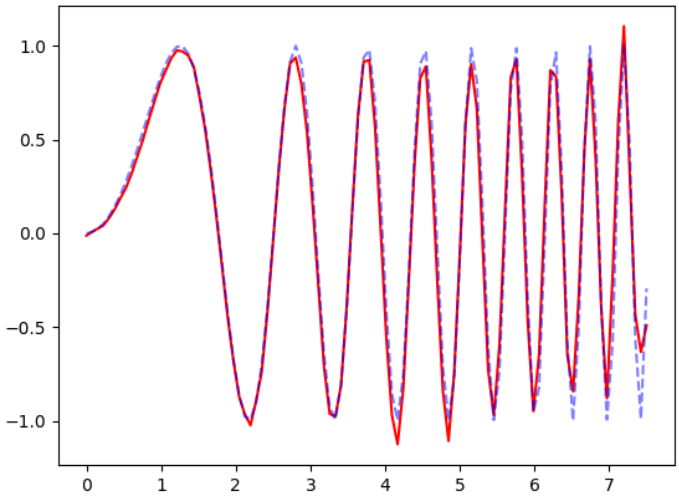
در این قسمت دو مورد را بررسی می‌کنیم: این که تاثیر افزایش تعداد نقاط ورودی برای تابع sin(x^2) در بازه‌ی ۰ تا ۷.۵ چقدر است و این که تاثیر کاهش تعداد نقاط ورودی برای همین تابع در بازه‌ی ۰ تا ۵ چقدر است.

در حالت اولیه ۱۰۰۰۰ نقطه داریم که ۸۰۰۰ تا برای آموزش و ۲۰۰۰ تا برای تست است. این تعداد را دو برابر می‌کنیم و تابع را در بازه‌ی ۰ تا ۷.۵ بررسی می‌کنیم:



MSE = 0.16105

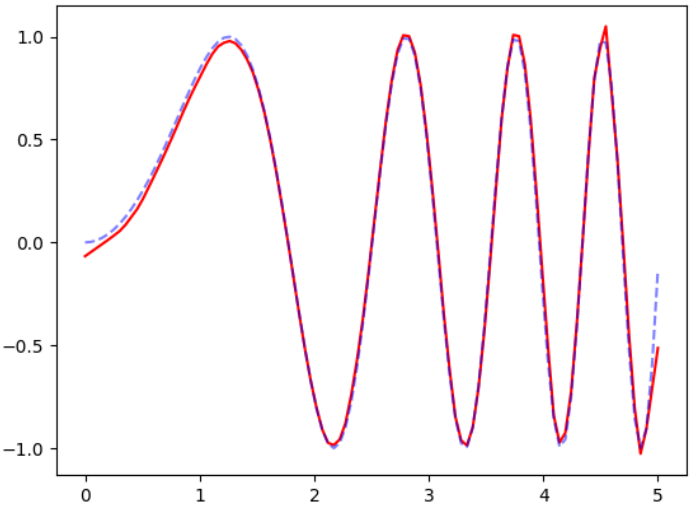
بنابراین قدری بهبود حاصل شده است، ولی مدل هنوز هم به خوبی عمل نمی‌کند. اگر این تعداد را یک و نیم برابر کنیم خواهیم داشت:



MSE = 0.01015

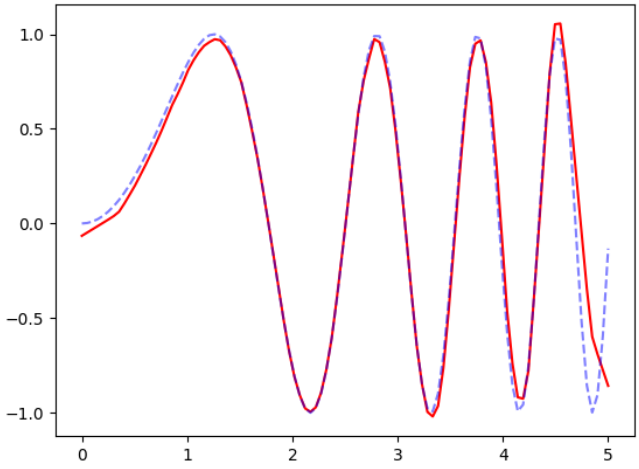
بنابراین مدل به دقت خوبی می‌رسد. چند نکته وجود دارد: مدل ما می‌توانست در بازه ۰ تا ۵ دقت خوبی داشته باشد، ولی با افزایش دامنه، ضعیف شده بود. با افزایش تعداد نقاط به نوعی (مطابق انتظارمان) با افزایش دامنه مقابله کردیم و مشکل ناشی از افزایش دامنه برطرف شد. اما نمی‌توان انتظار داشت که همواره با افزایش تعداد نقاط، مشکل حل شود، زیرا اولا مدل ما تعداد ثابتی epoch و batch size دارد و الزاما از افزایش نقاط بهره نمی‌برد، دوما مدل ما با توجه به نورون‌هایش تا حد ثابتی توانایی یادگیری پیچیدگی را دارد و قرار نیست با خوراندن نقاط بیشتر و افزایش epoch بتوانیم تابع سخت تری در آن بگنجانیم.

حال با ۵۰۰۰ نقطه بازه‌ی ۰ تا ۵ را تحلیل می‌کنیم:



MSE = 0.00175

بنابراین افت شدیدی نداشتیم. اگر ۳۰۰۰ نقطه استفاده کنیم:



MSE = 0.01089

بنابراین افت کیفیت مشاهده می‌شود. برای داشتن یک یادگیری خوب بهتر است برای بازه‌ی ۰ تا ۵ همان ۱۰۰۰۰ نقطه را داشته باشیم، ولی برای بازه‌ی ۰ تا ۷.۵ نیازمند ۳۰۰۰۰ نقطه هستیم تا خروجی حدودا همان کیفیت را داشته باشد.

میزان پیچیدگی تابع مورد نظر

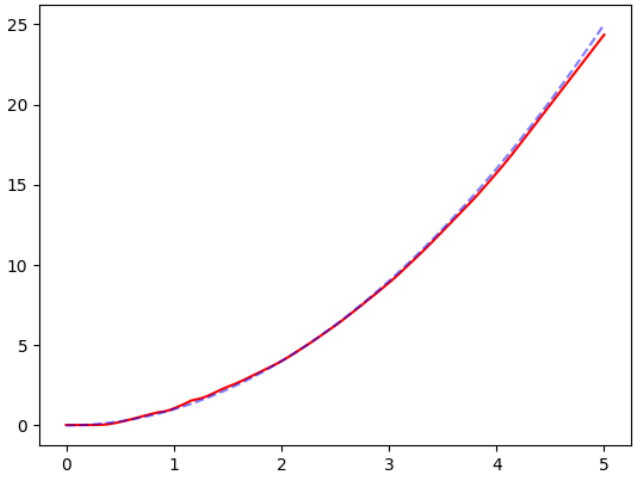
این موضوع به تفصیل در ابتدای این بخش بررسی شد. تابع پیچیده‌تر نیازمند شبکه‌ی پیچیده‌تری برای دریافت روند است، و برونیابی تابع پیچیده سخت تر است. یعنی اگر مثلا برای یک تابع در بازه‌ی کوچکی MSE = x حاصل شود، با افزایش بازه، همین مدل می‌تواند دقت خود را تا حد خوبی حفظ کند، ولی برای تابع پیچیده این امر صادق نیست، زیرا روند تابع ممکن است عوض شود و پیشبینی آن عملا غیرممکن است. به عنوان مثال، تابع x^2 همواره مشتق دوم ثابتی دارد و می‌توان با یک شبکه‌ی ساده آن را یاد گرفت و برونیابی هم کرد. ولی تابع sin(x^2) دارای بسامدی متغیر است و یادگیری آن حتی در یک بازه سخت است، و برون‌یابی آن با این روش‌ها نزدیک به غیرممکن. از آنجایی که می‌خواستیم مدل به چالش کشانده شود، در سرتاسر این بخش از سخت‌ترین تابعمان کمک گرفتیم.

بخش دوم: تابع یک‌بعدی با نویز

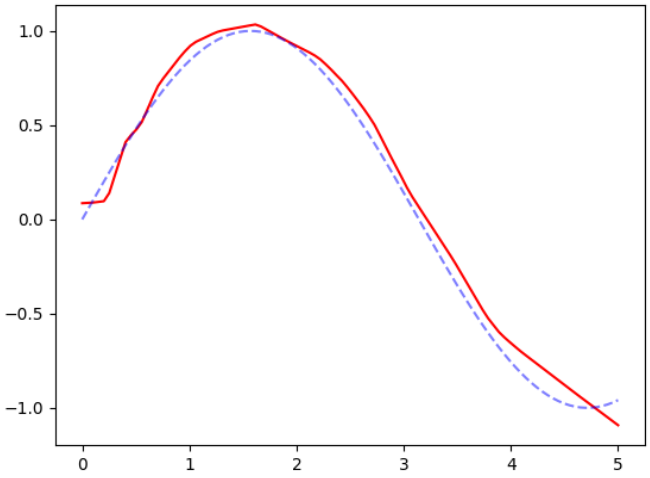
در این قسمت با استفاده از امکان نویز گوسی که در pointgen گذاشته بودیم (از پروژه‌ی ژنتیک مانده بود) استفاده می‌کنیم و روند یادگیری را مشاهده می‌کنیم. دقت داریم که چون توضیحات مفصل در قسمت اول آمده است، در این قسمت بیشتر به نتایج و مقایسه‌ی آنها می‌پردازیم و نه توضیح دادن این که چه اتفاقی افتاده است.

میزان پیچیدگی تابع مورد نظر

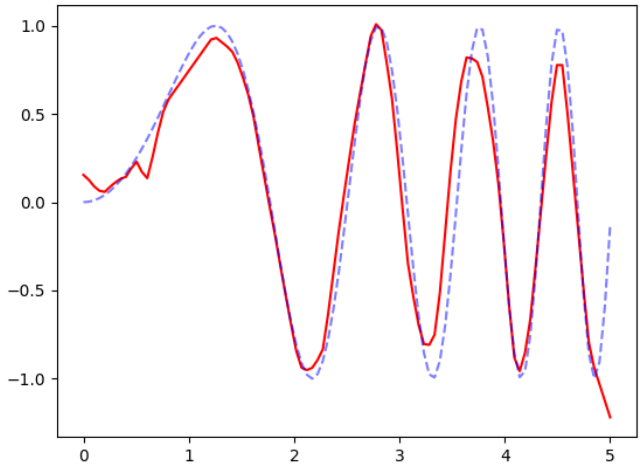
سه تابع x^2 و sin(x) و sin(x^2) را با ۱۰۰۰۰ نقطه و شبکه‌ی ثابت قسمت قبل یاد می‌گیریم. داریم:



MSE = 1.10679



MSE = 0.98490



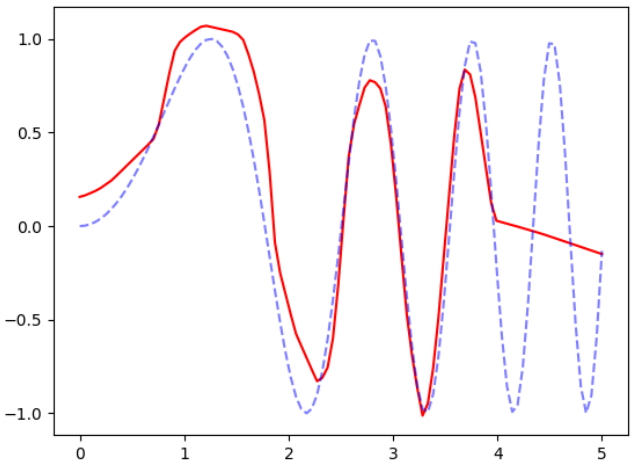
MSE = 0.99285

در حالت بدون نویز MSE ما باید در حدود 0.01 می‌بود، ولی الان در هر سه مدل به حدود 1 رسیده است. چیزی که جالب است این است که نویز ما gauss(0, 1) است!

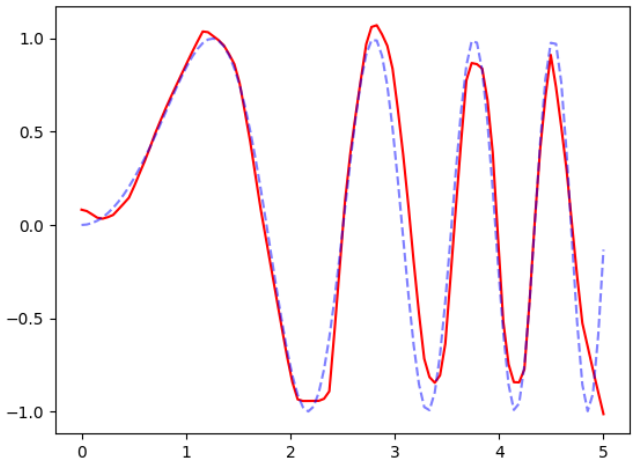
در حالتی که نویز داریم نمی‌توان به دنبال کاهش MSE کمتر از یک حدی بود، و این حد برای نویز گوسی ما همین 1 است. اگر MSE جایی کمتر از این مقدار شود یعنی عملا مدل روی نقاط ما overfit شده است. زمانی که نویز داریم مشکل overfit جدی‌تر است و باید مراقب باشیم که مدل ما خیلی به نقاط نویزی تن ندهد. به نظر می‌آید همین الان هم تا حدی این مشکل وجود دارد ولی خیلی جدی نیست.

تعداد نقاط ورودی

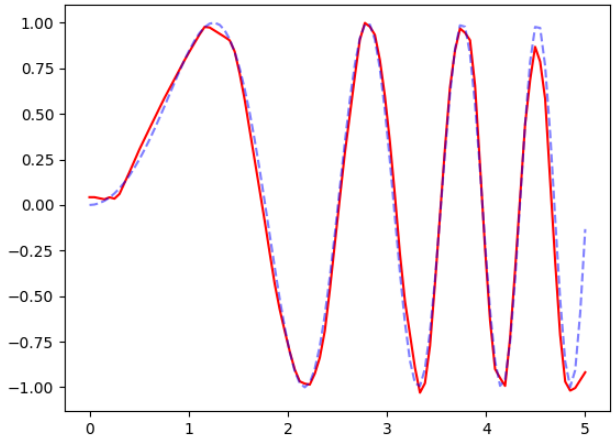
برای تابع sin(x^2) که در قسمت قبل به تفصیل بررسی شد، اینجا تحلیل تعداد نقاط ورودی را انجام می‌دهیم.
دیتاست را یک بار ۵۰۰۰ نقطه، یک بار ۱۰۰۰۰ نقطه و یک بار ۲۰۰۰۰ نقطه می‌گذاریم (به ترتیب).



MSE = 1.08705



MSE = 1.05288



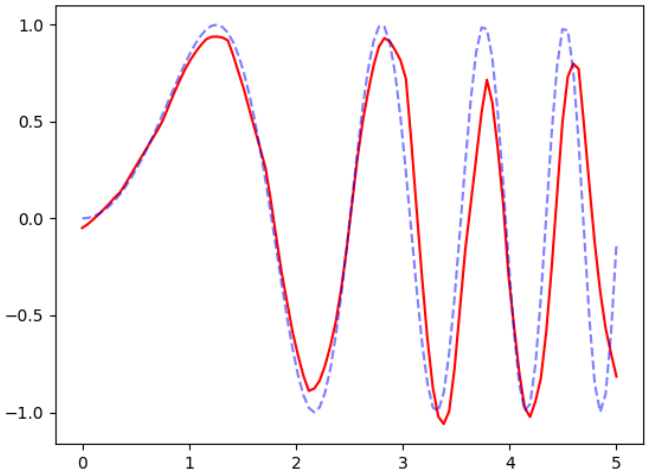
MSE = 1.03368

اگرچه MSE فرق چندانی نکرده است ولی ظاهر توابع بهتر شده. دقت داریم که چون برد تابع کوچک است، تغییر عجیبی در MSE نخواهیم داشت و بهبودهای ما هم تغییر کمی در MSE ایجاد می‌کنند، ولی ظاهر تابع بهتر شده است.

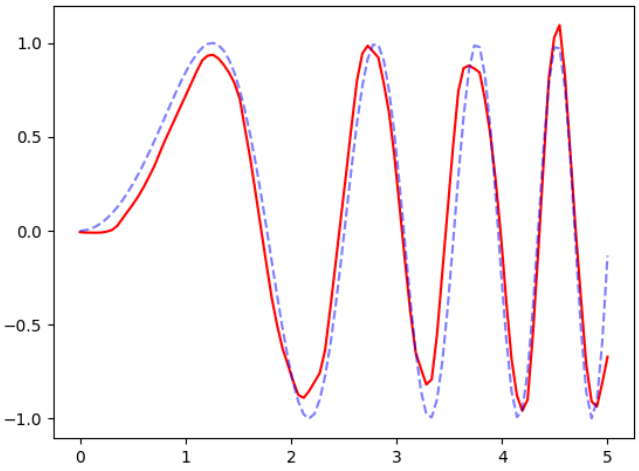
نکته حائز اهمیت این است که چون تابعمان ۱۰۰ تا epoch و هریک با ۱۰۰ نمونه انجام می‌دهد، در حال حاضر حداکثر استفاده از ۲۰۰۰۰ نقطه را نکرده‌ایم، بلکه با افزایش نقاط، بازنمایی دیتاست ما (representative بودن) بهتر شده است. اگر بخواهیم از این هم بهتر شود باید از نقاط زیادشده استفاده کنیم.

تعداد چرخه‌های شبکه برای تکمیل یادگیری

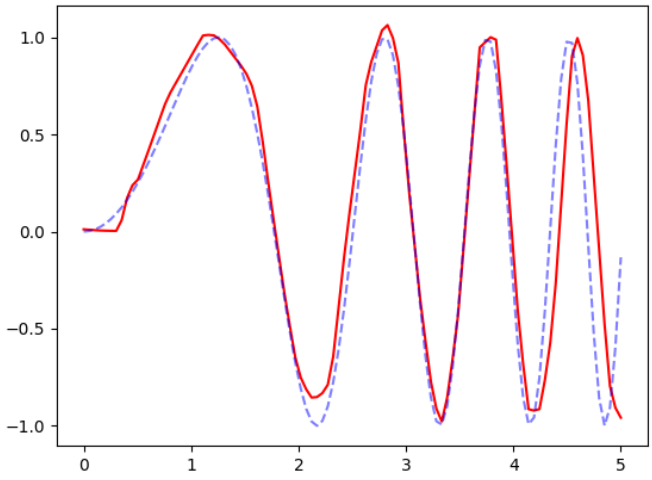
با دیتاست حاوی ۲۰۰۰۰ نقطه، تعداد چرخه‌های یادگیری را کم و زیاد می‌کنیم. از آنجایی که موضوع epoch و batch size در بخش اول به تفصیل مورد بررسی قرار گرفت، در این قسمت به تغییر epoch اکتفا می‌کنیم (زیرا عملا تغییر این دو تقریبا از یک جنس است) و نتایج حاصل از کم و زیاد کردن آن را مشاهده می‌کنیم. یادگیری به ترتیب با ۵۰، ۱۰۰ و ۲۰۰ مرحله انجام شد و نتیجه در ادامه آورده شده است:



MSE = 1.07754

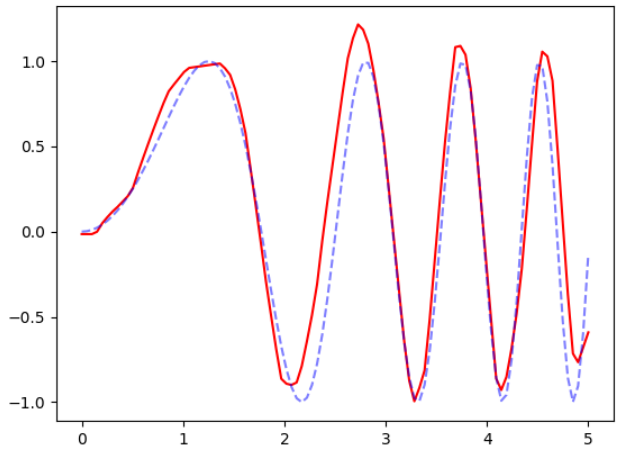


MSE = 1.02914



MSE = 1.05387

با وجود این که مراحل زیاد شد MSE زیاد شد، بنابراین به نظر می‌آید دچار overtrain شده‌ایم. اگر از ۱۵۰ مرحله استفاده کنیم خواهیم داشت:

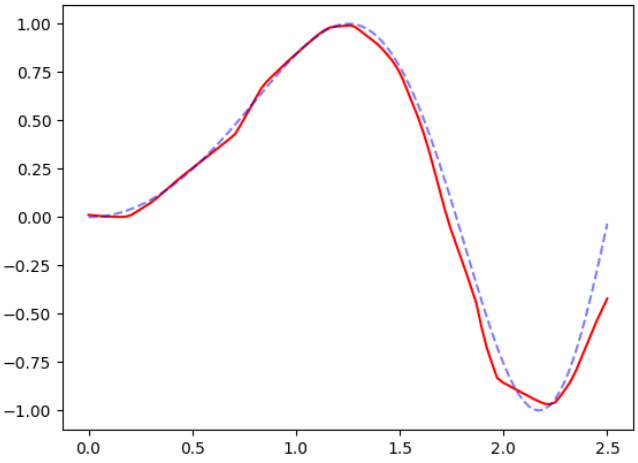


MSE = 1.04622

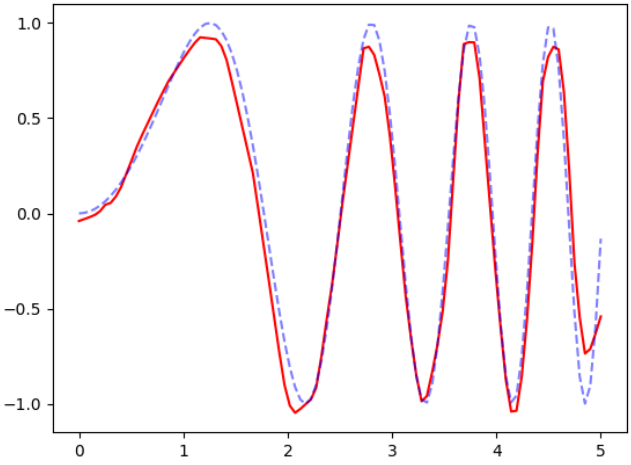
بنابراین جواب ما در حدود همان ۱۰۰ مرحله است، ولی بهتر است دیتاست ما بزرگتر باشد تا بازنمایی بهتر شود و نویز تاثیر کمتری بگذارد (در حالتی که نویز نبود ۱۰۰۰۰ نقطه هم نتیجه خوبی داشت ولی الان ۲۰۰۰۰ تا بهتر است).

وسعت دامنه‌ی ورودی

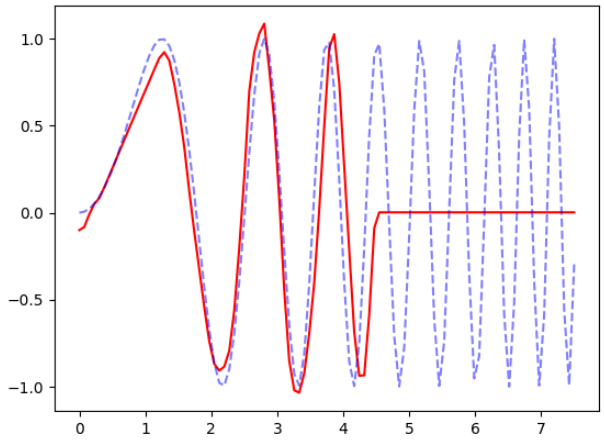
حال که به مدل خوبی رسیدیم سعی می‌کنیم با ریز و درشت کردن بازه، نتیجه را مشاهده کنیم. از آنجایی که نویز رندوم است، انتظار داریم در بازه‌ی کوچک، نتیجه به مدل بدون نویز نزدیک شود (زیرا نویز نقاط ممکن است همدیگر را کنسل کنند) ولی نتیجه در بازه‌ی بزرگ قطعا بد خواهد بود. نقطه‌ی شروع بازه ۰ است و نقطه‌ی انتها را ۲.۵ و ۵ و ۷.۵ می‌گذاریم (به ترتیب) و داریم:



MSE = 1.00340



MSE = 1.02637

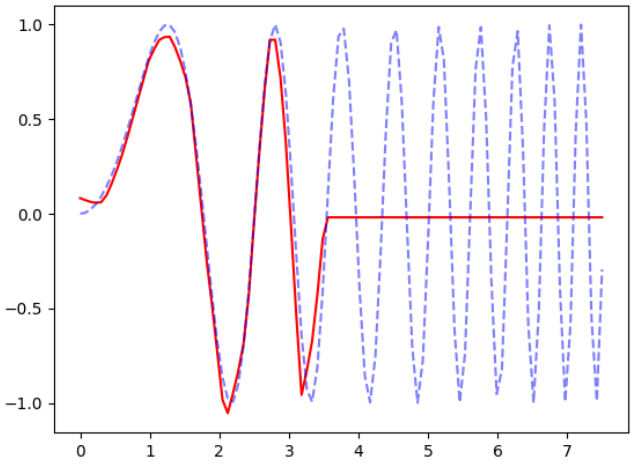


MSE = 1.23339

همانطور که انتظار داشتیم، یادگیری در زمانی که بازه بزرگ باشد سخت می‌شود، مخصوصا وقتی که تابعمان پیچیده است (مطابق توضیحات بخش اول).

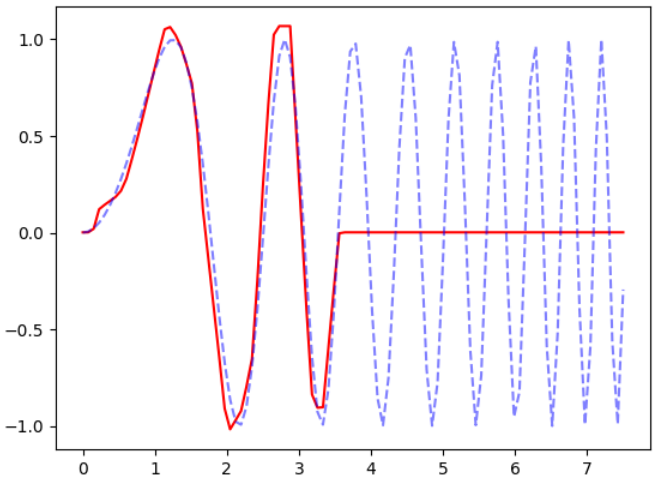
تعداد لایه‌ها و نورون‌های شبکه

در این قسمت هم به پیچیده‌تر شدن و هم به ساده‌تر شدن مدل می‌پردازیم. ابتدا سعی می‌کنیم مدل را کمی قوی‌تر کنیم تا شاید عملکرد در بازه‌ی ۰ تا ۷.۵ بهتر شود. یک لایه اضافه کرده و تعداد نورون‌ها در لایه‌های ۴۰۰ تایی را ۵۰۰ می‌کنیم. داریم:



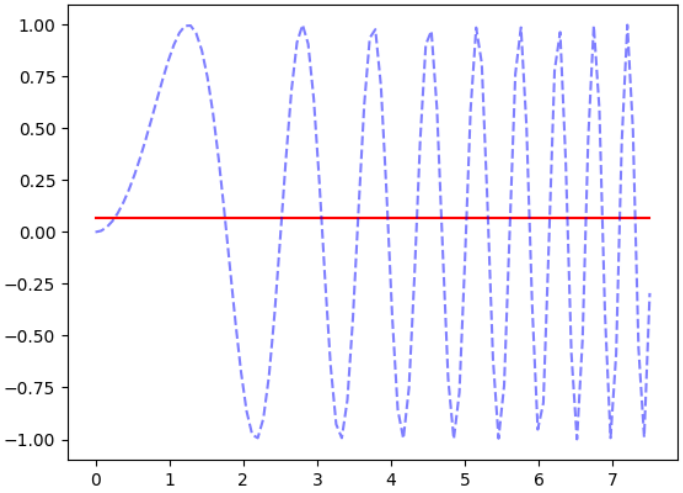
MSE = 1.26716

بنابراین مشاهده می‌شود هنوز مشکل هست و بدتر هم هست. باز هم شبکه را گسترش می‌دهیم تا ۷ لایه شود به ترتیب ۵۰۰ و ۱۰۰۰ و ۱۰۰۰ و ۵۰۰ و ۲۵۰ و ۱۰۰ و ۱ نورون. داریم:



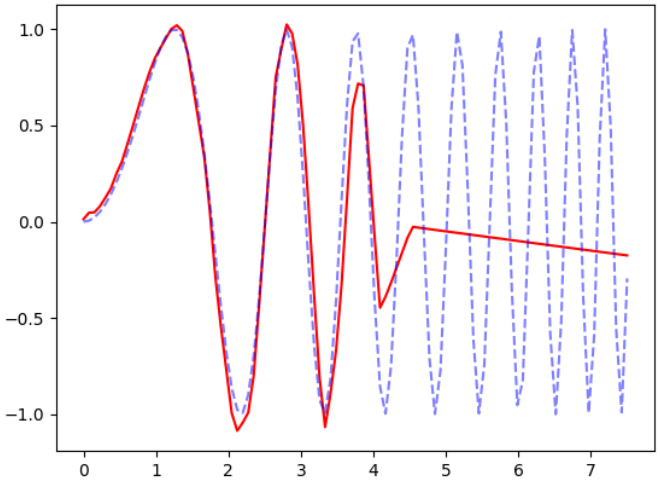
MSE = 1.25924

بنابراین پیچیده شدن شبکه به خودی خود مشکل را حل نمی‌کند. در کنار این کار اگر مراحل یادگیری را هم زیادتر کنیم داریم:



MSE = 1.45003

بنابراین آنچه مشاهده می‌شود این است که الزاما با پیچیده شدن شبکه، وضع بهتر نمی‌شود. این موضوع وقتی نویز داریم شدت بیشتری هم پیدا می‌کند زیرا تاثیر نویزها با پیچیده شدن شبکه بیشتر نمایان می‌شود و مدل overtrain می‌شود و برای کاهش MSE خود را ناچار می‌بیند که یک خط صاف برود. اگر شبکه را ساده کنیم (از حالت اول هم ساده‌تر) تا به ۲۵۰، ۴۰۰، ۲۰۰، ۷۰ و ۱ نورون برسیم، داریم:



MSE = 1.26898

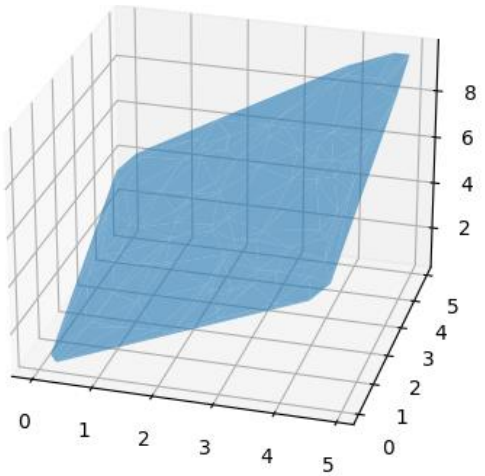
بنابراین آنچه در اول کار داشتیم مناسب بود و نه زیادی پیچیده بود و نه زیادی ساده. همانطور که اول کار گفته شد وقتی نویز با توزیع gauss(0, 1) داریم رسیدن به MSE=1 عملا بهترین چیزی است که می‌توانیم بخواهیم.

بخش سوم: توابع با ابعاد بالاتر

این قسمت نیز مانند قسمت‌های قبل است و فقط با تفاوت‌های کمی، بجای یک ستون ورودی، چند ستون ورودی داریم. تنها تفاوت قابل توجه این است که امکان نمایش دو تابع به صورت خوبی وجود ندارد – اولا فقط می‌توان توابع دو بعدی را به صورت قابل درک نمایش داد، دوما که نمایش تابع ایجاد شده در کنار تابع مورد انتظار، تصویر را شلوغ می‌کند و درک تفاوت‌های اندکی که میان این دو هست عملا غیرممکن خواهد بود.
بنابراین در ابتدا چند تصویر از توابع ایجاد شده نشان می‌دهیم و در ادامه از MSE برای درک کیفیت استفاده می‌کنیم.

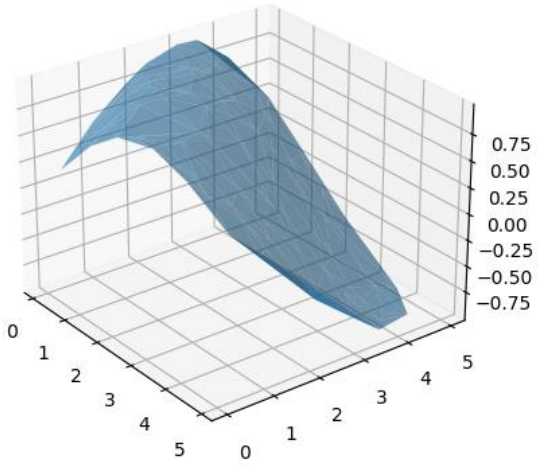
در ابتدا شبکه‌ی ما دارای ۴۰۰ و ۴۰۰ و ۴۰۰ و ۱۰۰ و ۱ نورون است و در ۱۰۰ مرحله با بسته‌های ۱۰۰ تایی آموزش می‌بیند.

به عنوان مثال، تابع ایجاد شده برای f = x + y با دامنه‌ی ۰ تا ۵ برای هر دو ورودی برابر است با:



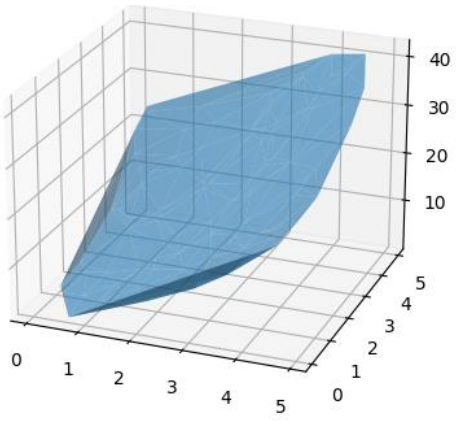
MSE = 0.02562

یا تابع ایجاد شده برای f = sin(x/2 + y/2) با همان دامنه برابر است با:



MSE = 1.51520

و یا تابع ایجاد شده برای f = xˆ2 + yˆ2 با همان دامنه برابر است با:



MSE = 0.00114

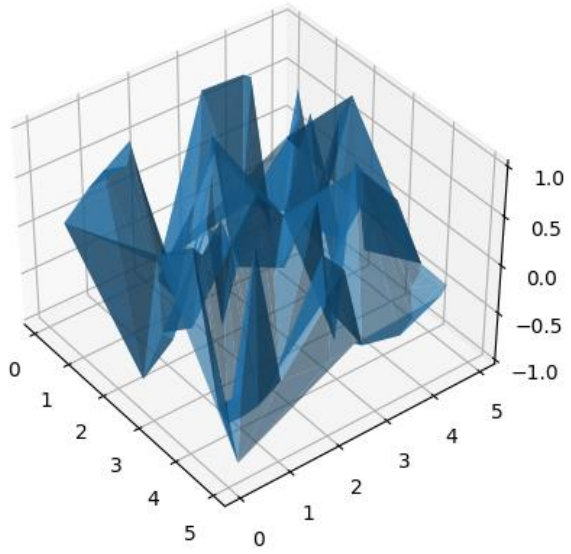
با توجه به اینها مشاهده می‌شود که پیشبینی توابع چندبعدی هم کار عجیبی نیست. در ادامه تحلیل‌هایی که در دو بخش قبل داشتیم را دوباره تکرار می‌کنیم، ولی برای بررسی تابع ایجاد شده، به MSE بسنده می‌کنیم.

تعداد نقاط ورودی

تابع sin(x/2 + y/2) با ۵۰۰۰ و ۱۰۰۰۰ و ۲۰۰۰۰ نقطه به ترتیب دارای MSE های 0.00047 و 6.75648 و 3.15093 است. از آنجایی که نقاط تصادفی هستند به نظر می‌آید زیاد شدن نقاط (بیش از حد لازم) باعث می‌شود تابع روی قسمتی از ورودی overfit شود و در کل ضعیف شود. بنابراین بهتر است تعداد نقاط را الکی زیاد نکنیم.

میزان پیچیدگی تابع مورد نظر

با ۱۰۰۰۰ نقطه سه تابع x+y و sin(x+y) و sin(x^2 + y^2) را بررسی می‌کنیم. MSE به ترتیب برابر است با 1.09294 و 7.13017e-05 و 0.04457 است. تابع آخر از آنجایی که جالب است اینجا آورده شده است:



مشاهده می‌شود که overfit شدیدی رخ داده است که MSE را خیلی کم کرده است. دقت داریم که تابع ثابت f=0 برای این موارد منجر به MSE=1 خواهد شد، پس وقتی به MSE=1 می‌رسیم یعنی یک جای کار اساسی می‌لنگد و شبکه کار خاصی برایمان نکرده است.

تعداد لایه‌های شبکه و نورون‌های هر لایه

شبکه در حالت فعلی: ۴۰۰ و ۴۰۰ و ۴۰۰ و ۱۰۰ و ۱ نورون

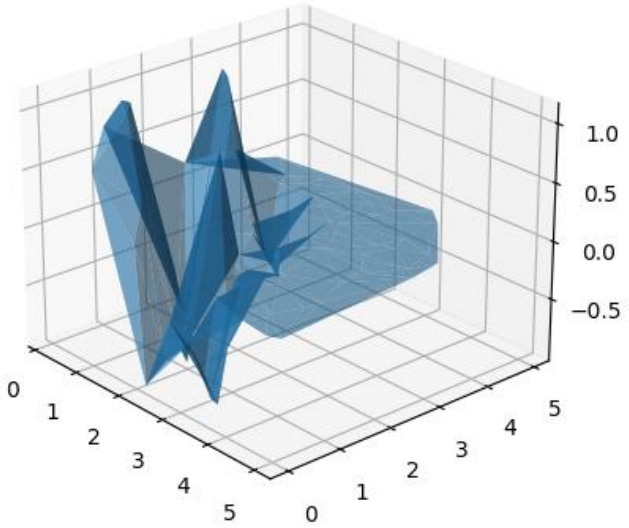
شبکه در حالت دوم: ۲۵۰ و ۲۵۰ و ۲۵۰ و ۱۰۰ و ۱ نورون

شبکه در حالت سوم: ۴۰۰ و ۶۰۰ و ۷۰۰ و ۲۰۰ و ۱۰۰ و ۱ نورون

به ترتیب MSE برای ۱۰۰۰۰ نقطه از تابع sin(x^2 + y^2) بدست می‌آید:

0.06264 و 0.04918 و 0.10021

چون با توجه به اعداد و شکل حاصل به نظر مشکل از جنس overfitting است حالت ۲۰۰ و ۲۰۰ و ۵۰ و ۱ نورون را امتحان می‌کنیم:



MSE = 0.30480

بنابراین کماکان فشار زیادی به شبکه وارد شده و حاصل خاصی نداریم. البته با توجه به شکل واقعی این تابع، انتظار بالایی هم نداریم، ولی به دنبال این هستیم که MSE کاهش یابد نه افزایش.

تعداد چرخه‌های شبکه برای یادگیری

این بار ۲۵۰ و ۴۰۰ و ۴۰۰ و ۱۰۰ و ۱ نورون داریم و تابع sin(x+y) را بررسی می‌کنیم.

حالت اول: epoch = 50, batch size = 50

حالت دوم: epoch = 100, batch size = 100

حالت سوم: epoch = 150, batch size = 150

تعداد نقاط را ۲۰۰۰۰ گذاشته تا تفاوت محسوس باشد. به ترتیب MSE به دست آمده برابر است با 0.00012 و 7.39693e-05 و 3.33311e-05 و با توجه به تصویر توابع، به نظر می‌آید پیشرفت حاصل شده است. بنابراین در وضعیت overfit نیستیم و وقتی نقطه هست می‌توان از آن استفاده کرد تا دقت بهتر شود.

وسعت دامنه‌ی ورودی

برای تابع زیربخش قبل، تحلیلمان را با همان ۲۰۰۰۰ نقطه و یادگیری حالت دوم، با دامنه‌ی ۰ تا ۵، ۰ تا ۷.۵ و ۰ تا ۱۰ انجام می‌دهیم و نتیجه را مشاهده می‌کنیم. MSE به ترتیب برابر است با 6.75793e-05 و 0.00023 و 0.00151.

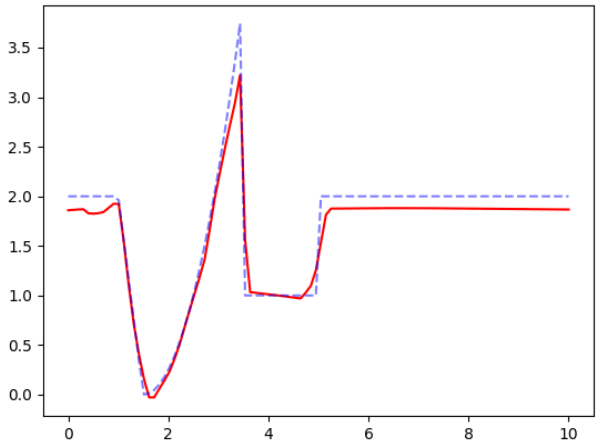
واضح است که چرا این اتفاق می‌افتد و این موضوع در دو بخش قبل به تفصیل بررسی شد، وقتی بازه بزرگتر می‌شود عملا حجم چیزهایی که مدل باید یاد بگیرد زیادتر خواهد شد و یک مدل ثابت با تعداد نقاط و ... ثابت، در طی این امر، نتیجه‌ی ضعیف‌تری خواهد داشت.

بخش چهارم: یادگیری داده‌های نامنظم

در این بخش مطابق خواسته‌ی تمرین، تابعی بدون نظم برای نقاط یادگیری در نظر می‌گیریم. برای این منظور عملا مجبوریم تابعی چندضابطه‌ای ایجاد کنیم. مجددا کار خاصی انجام نداده‌ایم و کدمان همان کد بخش اول و دوم است، با این تفاوت که تابع عجیب و غریبی برای تولید نقاط استفاده شده است. برای پیاده کردن این تابع از np.piecewise استفاده شده است:

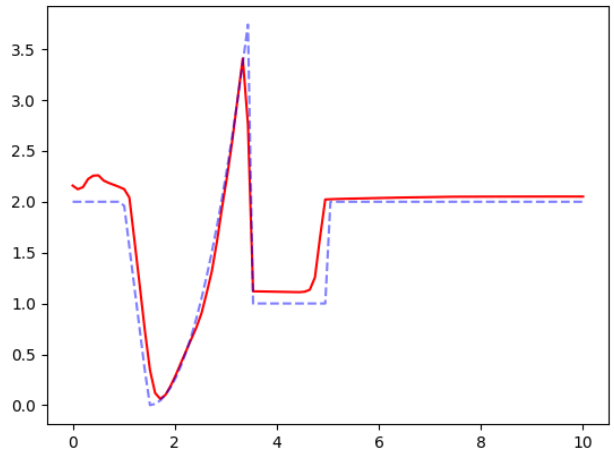
```
return np.piecewise(x, [(x > 1) & (x < 1.5),
                        (x >= 1.5) & (x < 3.5),
                        (x >= 3.5) & (x < 5),
                        (x <= 1) | (x >= 5)],
                        [lambda x: 6 - 4 * x,
                        lambda x: (x - 1.5)**2,
                        1, 2]) + noise
```

و برای ۱۰۰۰۰ نقطه در بازه‌ی ۰ تا ۱۰، با ۱۰۰ دور آموزشی هریک با ۵۰ داده، روی شبکه‌ای با ۴۰۰ و ۴۰۰ و ۴۰۰ و ۱۰۰ و ۱ نورون، داریم:



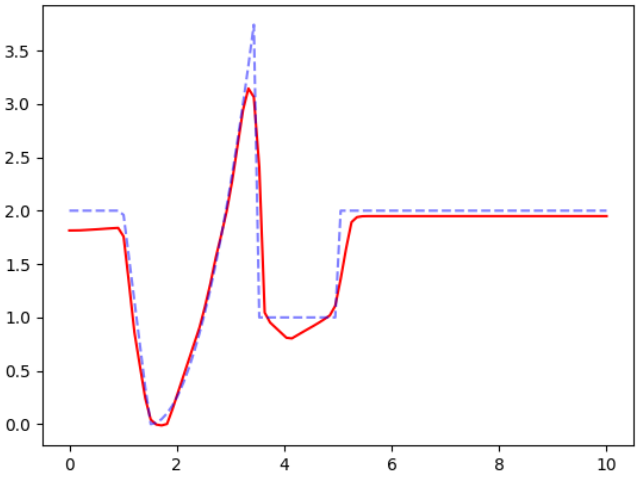
MSE = 1.05515

دقت داریم که تابع ناپیوستگی یا به اصطلاح «پرش» دارد، و با این وجود، شبکه آن را تا حد خوبی یاد گرفته است. سعی می‌کنیم مقداری بهتر شود، برای این منظور، ۱۵۰ دور آموزشی هریک با ۱۰۰ داده را امتحان می‌کنیم:



MSE = 1.09374

به نظر می‌آید مدل در حال overfit شدن است، بنابراین همان روند یادگیری قبلی را انجام می‌دهیم ولی تعداد نورون سه لایه اول را به ۳۰۰ کاهش می‌دهیم. داریم:



MSE = 1.02006

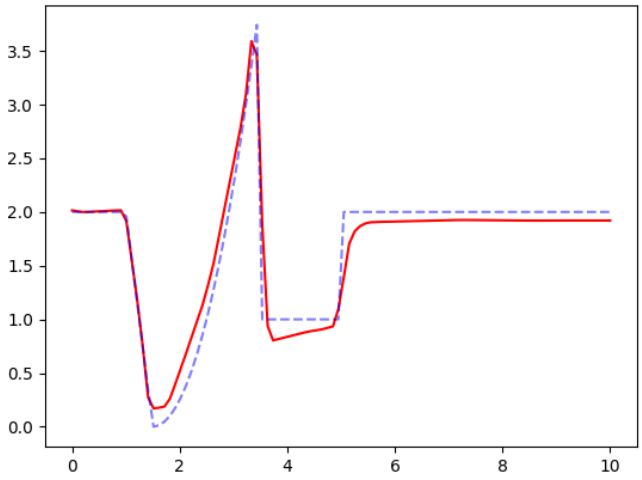
بنابراین MSE کم شده است ولی نتیجه‌ی زیبایی نداریم. به نظر می‌آید تابعی که در ابتدا استفاده کردیم — که گل سرسبد بخش اول پروژه بود — بهترین تعادل را برایمان ایجاد می‌کند. البته به چند مورد باید دقت داشت:

اولا که می‌توان یادگیری زیادتری انجام داد و شبکه‌ی ساده‌تری استفاده کرد. در این حالت احتمالا بتوانیم در زمان بیشتری به همین جواب برسیم ولی منابع کمتری هدر بدهیم.

دوما که تعادل میان تابع (از نظر شباهت) و MSE لازم است، زیرا اگر فقط MSE کم شود و تابع حالت نافرمی به خودش بگیرد، ممکن است در دنیای واقعی نتیجه‌ی خوبی نداشته باشد.

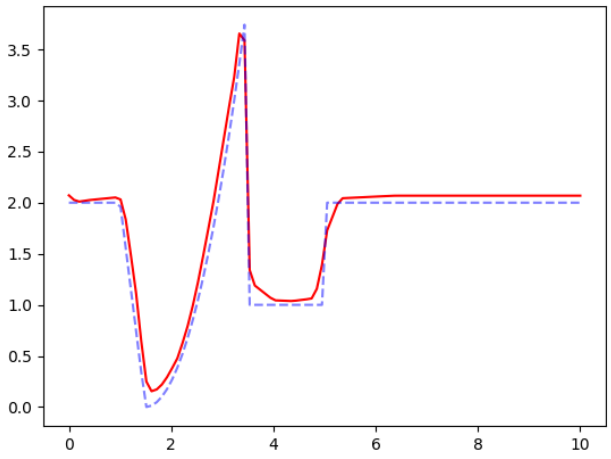
سوما توجه داریم که بخاطر پرش های تابع و نقاط نوک تیز و حرکات با بسامد بالا، یادگیری چنین تابعی ذاتا سخت است.

به نظر می‌آید بهترین راه برای بهبود نتیجه، افزایش تعداد نقاط باشد. تعداد نقاط را به ۲۰۰۰۰ رسانده و داریم:



MSE = 1.02108

با افزایش دور یادگیری به ۱۲۰ و تعداد نقاط هر دور به ۱۰۰ داریم:



MSE = 0.96324

که به نظر نتیجه‌ی خوبی می‌آید – هم تابع ایجاد شده شبیه خطچین است و هم MSE به زیر ۱ رسیده است.

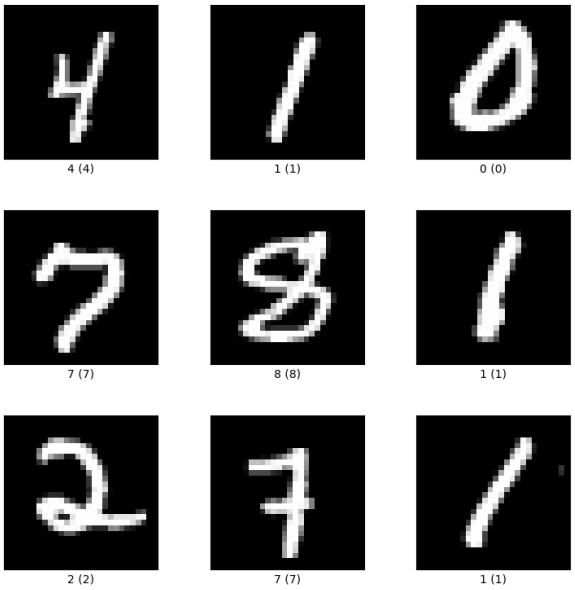
اگر پرش ناگهانی داشته باشیم کار مدل سختتر خواهد شد ولی اصلا نشدنی نیست (بالاخره شبکه عصبی است، نه رگرسیون ساده!). از آنجایی که هرچقدر تابع پیچیده‌تر باشد شبکه‌ی مورد نیاز بزرگتر است، اندازه‌ی شبکه معادل خشم دانشجو است.

بخش پنجم: تشخیص رقم در MNIST

در این قسمت یک کد کاملا نو استفاده شده است که روی دیتاست MNIST (و برای برخی قسمت‌ها Fashion MNIST) یادگیری انجام می‌دهد. برای اجرای مستقیم کدهای این قسمت، چنانچه قبلا keras.datasets.mnist دریافت نشده باشد، یک راهکار برای دور زدن تحریم گوگل لازم است (بنده از shecan استفاده کردم).

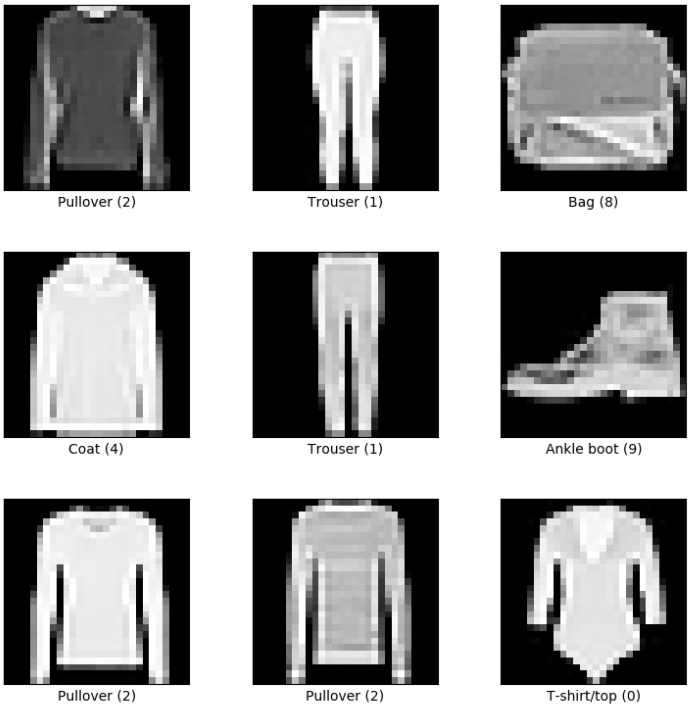
بنده در ابتدا میان MNIST و CIFAR10 دودل بودم، زیرا هردو ده کلاسه هستند و تصاویر کوچکی دارند که باعث می‌شود نیازی به شبکه‌های خاص نداشته باشیم و بتوان با زمان معقول به یک مدل خوب رسید. علت انتخاب MNIST این بود که اولا رایج‌تر و به نوعی «ملاک» تر است و نقطه‌ی ورود عموم مردم به شبکه‌های عصبی است. همچنین تصور بنده این بود که یادگیری روی این دیتاست‌ها نیازمند زمان زیادی از مرتبه‌ی ساعت است،

بنابراین MNIST که حجم کمتری داشت را ترجیح دادم، ولی بعد از اولین یادگیری مشاهده کردم که سرعت یادگیری این دیتاست‌ها از یادگیری‌های چهار بخش قبل به مراتب کمتر است!



چند نمونه از داده‌های MNIST

از آنجایی که با یک شبکه می‌توان به سادگی دقت بالایی برای MNIST به دست آورد، برخی جاها از Fashion MNIST استفاده شده است که بتوان افت و خیز دقت را بیشتر مشاهده کرد.



چند نمونه از داده‌های این Fashion MNIST

توضیح کد

```
# load dataset from keras
(input_train, output_train), (input_test, output_test) = mnist.load_data()

# reshape data into relu-friendly numbers
input_train = input_train.reshape((input_train.shape[0], 28*28)).astype('float32')/255
input_test = input_test.reshape((input_test.shape[0], 28*28)).astype('float32')/255
output_train = to_categorical(output_train, 10)
output_test = to_categorical(output_test, 10)
```

در این قسمت، دیتاست از موجودی keras بارگذاری می‌شود. اگر این کار قبلا انجام نشده باشد لازم است که این دیتاست دانلود شود. در ادامه، فرمت تصاویر و خروجی‌ها تغییر می‌کند تا مناسب کار ما باشد. منظور این

است که هر تصویر را به شکل ۲۸ در ۲۸ پیکسل در می‌آوریم. از آنجایی که شبکه‌ی عصبی ReLU داریم (و در کل برای سایر نورون‌های رایج در کلاس‌بندی) بهتر است مقادیر پیکسل‌ها عدد کوچکتری باشد، برای همین تقسیم بر ۲۵۵ می‌کنیم (تا عددی در بازه‌ی ۰ تا ۱ بدست بیاید). خروجی‌ها هم برای کار کلاس‌بندی categorical شده‌اند.

```
15  # create model
16  model = Sequential()
17  model.add(Dense(32, input_dim = 28 * 28, activation= 'relu'))
18  model.add(Dense(64, activation = 'relu'))
19  model.add(Dense(10, activation = 'softmax'))
20
21  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
22
23  model.fit(input_train, output_train, batch_size=128, epochs=10, validation_split=0.1)
```

حال مشابه بخش‌های قبلی پروژه، شبکه عصبی تعریف می‌شود. اینجا چون کلاس‌بندی داریم و نه رگرسیون، تابع نورون‌ها را ReLU قرار داده و در نهایت یک لایه‌ی softmax داریم که تعیین کند کلاس infer شده کدام است.

سپس برای کامپایل کردن مدل، مجددا چون کلاس‌بندی داریم و نه رگرسیون، از تابع loss متفاوتی نسبت به بخش اول استفاده می‌کنیم و سنجه‌ی ما برای مدل «دقت» است و نه MSE.

```
25  # save on disk
26  model.save('p2_mnist')
27
28  # evaluate model
29  score = model.evaluate(input_test, output_test, verbose=0)
30
31  print("Test Loss: ", score[0])
32  print("Test Accuracy: ", score[1]*100, '%')
```

در این قسمت مدل ذخیره می‌شود (زیرا تصور اولیه بنده این بود که یادگیری آن خیلی طولانی است، راهی پیدا کردم که ذخیره شود و اگر دوباره به آن نیازی شد آماده باشد) و دقت آن اندازه‌گیری می‌شود.

لازم به ذکر است که دیتاست MNIST از اول split شده است و لزوما کار جالبی نیست که split آن را به هم بزنیم، زیرا ارزیابی مدل با همین split نوعی استاندارد است. بنابراین cross-validation هم عملا انجام نمی‌دهیم، ولی با یک split مشخص، حالات مختلف مدل را تست می‌کنیم تا نتیجه‌ای در همان حدود حاصل شود.

مشابه قسمت اول، دقت مدل‌های مختلف را از جنبه‌های مختلف بررسی می‌کنیم. از آنجایی که عملیات رگرسیون نیست عملا امکان ارزیابی چشمی مدل وجود ندارد، صرفا می‌توانیم با عددی که برای دقت بدست می‌آید مدل را قضاوت کنیم.

مدل اولیه – معادل با کدی که دیده شد – دارای دقت 96.870% است.

```
Test Loss:  0.10876163095235825
Test Accuracy:  96.86999917030334 %
```

تعداد نقاط ورودی

دیتاست ما به طور پیشفرض ۶۰۰۰۰ تصویر آموزشی دارد. در حالت عادی تمامی آنها را داریم پس گزینه‌ی پیش روی ما این است که بخشی از آن را دور بریزیم. بدیهی است که نتیجه‌ی این کار مثبت نیست ولی می‌توانیم شدت بد بودن این کار را ارزیابی کنیم.

چنانچه از ۱۰۰۰۰ تصویر آموزشی استفاده کنیم دقت به 92.790% می‌رسد. بدتر از ۶۰۰۰۰ تصویر است ولی خیلی هم بد نیست.

با استفاده از ۳۰۰۰۰ تصویر یعنی نصف دیتاست دقت به 95.820% می‌رسد که خیلی نزدیک به حالتی است که کل دیتاست استفاده شود.

با استفاده از ۵۰۰۰۰ تصویر یعنی بیشتر دیتاست دقت به 96.430% می‌رسد که کمتر از نیم درصد با حالتی که کل دیتاست استفاده شود فرق دارد.

بنابراین تاثیر کاهش تعداد اعضای آموزشی فجیع نیست. اما باید به چند مورد توجه داشت، اولا که مدل کنونی ما ۱۰ دور آموزشی دارد با سایز batch برابر ۱۲۸. پس تاثیر کاهش و افزایش دیتاست در اوردر چند ده هزار تاثیر شگرفی بجز تغییر بازنمایی دیتاست ندارد و تاثیر خودش را تمام و کمال نشان نمی‌دهد.

دوما که دیتاست ما دیتاست «آسانی» است چون شباهت عناصر زیاد است و اهمیت تصاویر و هر بعد ورودی (عملا پیکسل) یکنواخت است. در دیتاست‌هایی که به نوعی anomaly داریم اهمیت گستردگی دیتاست بیشتر است و خودش را بیشتر نشان خواهد داد.

در اینجا چون با چندصد نمونه به دقت حدود ۹۷ درصد رسیده بودیم، نمی‌توان تاثیر تغییر دیتاست را نشان داد (حتی اگر تعداد دور آموزشی را زیاد کنیم و ریسک overfit شدن هم داشته باشیم و هر کاری هم بکنیم، دقت نهایتا ۳ درصد می‌تواند زیادتر شود!).

میزان پیچیدگی

در اینجا میزان پیچیدگی مانند بخش‌های ابتدایی معنایی ندارد زیرا دیتاست ما ثابت است، ولی همانطور که انتظار داریم و در قسمت‌های اول دیدیم، اگر پارامترهایمان را ثابت نگه داریم (مشخصات شبکه و تعداد نقاط و ...) بدیهی است که با پیچیده شدن کار ما، دقت کم خواهد شد. واقعیتی که وجود دارد دیتاست‌های مد نظر ساده هستند (و خب این اولین تجربه‌ی بنده است) و بدون چالش خاصی دقتمان بالای ۹۰ درصد خواهد بود (برای دیتاست MNIST رسیدن به این دقت خیلی آسان است). برای حل این مشکل، دیتاست‌های متفاوتی ارائه شده‌اند (مانند Fashion MNIST برای دسته‌بندی میان ۱۰ دسته پوشاک) که رسیدن به دقت بالا نیازمند مقداری تدبیر بیشتر باشد، پس اگر دیتاست را جایگزین کنیم با همین شبکه دقتمان کم خواهد شد.

از آنجایی که دیتاست Fashion MNIST از لحاظ ابعاد و فرمت تصاویر خیلی شبیه به MNIST است و در کتابخانه آماده موجود است، کافیست بجای import کردن MNIST، Fashion_MNIST را قرار بدهیم. مابقی کد تغییری نمی‌کند.

```
Test Loss:  0.36441490054130554
Test Accuracy:  86.69999837875366 %
```

مشاهده می‌شود که دقت شبکه‌ای که روی MNIST حدود ۹۷ درصد بود، برای Fashion MNIST حدود ۸۷ درصد است یعنی ۱۰ درصد کمتر.

تعداد لایه‌های شبکه و تعداد نورون‌های هر لایه

برای دیدن این که شبکه‌ی ضعیف‌تر چه خواهد کرد، تعداد نورون‌ها را از ۳۲ و ۶۴ به ۲۰ و ۵۰ کاهش می‌دهیم و این را روی mnist بررسی می‌کنیم. دقت به 96.100% کاهش می‌یابد که اصلا کاهش چشمگیری نیست. اگر تعداد را به ۱۶ و ۳۲ کاهش بدهیم و مجددا یادگیری انجام بدهیم دقت به 95.390% کاهش می‌یابد که هنوز هم خیلی زیاد است. برای مقایسه، همین مدل روی fashion mnist دارای دقت 85.290% است.

نکته‌ی حائز اهمیت این است که افت دقت خیلی کم است، یعنی اگرچه با کوچک کردن شبکه دقت کم می‌شود و با انتخاب دیتاست سخت‌تر حتی کمترتر هم می‌شود، ولی افت دقت با کاهش چند نورون شدید نیست که به نوعی بیانگر این است که تفکیک کردن انواع داده‌هایمان ساده است (ارقام تفاوت قابل ملاحظه‌ای با همدیگر دارند). اگر بخواهیم بیشتر از اینها تعداد نورون‌ها را کم کنیم شبکه تمام می‌شود.

حال سعی می‌کنیم با پیچیده کردن شبکه، دقت روی fashion mnist را بالا ببریم. تعداد نورون‌ها را ۵۰ و ۱۰۰ قرار می‌دهیم و یادگیری را تکرار می‌کنیم.

```
Test Loss:  0.3572291135787964
Test Accuracy:  87.41999864578247 %
```

بنابراین دقت بهتر می‌شود (ولی نه خیلی). حال یک لایه با ۷۵ نورون میان این دو اضافه می‌کنیم و داریم:

```
Test Loss:  0.3452637791633606
Test Accuracy:  87.87000179290771 %
```

اگر تعداد نورون‌ها را به ۶۰ و ۱۰۰ و ۱۲۰ برسانیم خواهیم داشت:

```
Test Loss:  0.34538787603378296
Test Accuracy:  87.95999884605408 %
```

بنابراین به نظر می‌آید که به نوعی به ته خط رسیده‌ایم و با پیچیده کردن شبکه نمی‌توان به سادگی به دقت ۹۷ درصدی که در دیتاست ساده‌تر داشتیم برسیم.

تعداد چرخه‌های شبکه برای تکمیل یادگیری

در این قسمت بررسی می‌کنیم که تعداد چرخه‌ها چه تاثیری روی یادگیری ما خواهند داشت. ابتدا روی دیتاست mnist با شبکه‌ی ۳۲ و ۶۴ نورونی، تعداد داده‌های هر چرخه را از ۱۲۸ به ۹۰ کاهش می‌دهیم. دقت به 96.450% می‌رسد که یعنی افت شدیدی نداشتیم. اگر به ۶۰ کاهش بدهیم دقت به 96.840% می‌رسد که حتی از حالت اولیه هم زیادتر است (گرچه این تغییرات دهم درصدی شانسی هستند عمدتا و معنای خاصی ندارند).

اگر با همان ۹۰ داده برای هر دوره، تعداد دوره را از ۱۰ به ۸ کاهش بدهیم دقت می‌شود 96.450% و اگر به ۵ دوره کاهش بدهیم به 96.280% می‌رسیم. اگر حتی باز هم آن را کم کنیم و ۳ دور یادگیری داشته باشیم دقت می‌شود 95.550% . نکته این است که احتمالا در دورهای ۹ و ۱۰ تغییر خاصی نداریم، یعنی مدل ما مقدار خیلی کمی overfit می‌شود. همانطور که گفته شد تغییرات از مرتبه دهم درصد خیلی معنادار نیستند، زیرا خیلی اوقات ممکن است با کاهش یادگیری، بخش train دیتاست به test شبیه‌تر شود و دقت بالاتر به نظر بیاید، ولی این چیز خوبی نیست. بنابراین به نظر می‌آید ۸ الی ۱۰ دور یادگیری و حدود ۹۰ الی ۱۰۰ داده در هر دور، برای mnist کفایت کند.

در ادامه سعی می‌کنیم با افزایش یادگیری، روی شبکه‌ای با ۵۰ و ۱۰۰ نورون، دقت روی fashion mnist را زیادتر کنیم. به یاد داریم دقت چنین شبکه‌ای با ۱۰ دور یادگیری و ۱۲۸ داده در هر دور حدود ۸۷ درصد بود.

با افزایش تعداد دور به ۱۲ دقت می‌شود 87.480% که فرق خاصی نکرده است. با افزایش تعداد دور به ۱۵ دقت می‌شود 87.320% که نتیجه‌ی خوبی نیست.

بنابراین با افزایش دور نمی‌توان به سادگی این مشکل را حل کرد. اگر با همان ۱۲ دور تعداد داده‌ی هر دور را به ۲۰۰ برسانیم دقت می‌شود 88.020% که بالاترین عددی است که تا کنون داشته‌ایم. اگر حالا تعداد دور به ۱۵ برسد دقت می‌شود 87.680% که خبر خوبی نیست. بنابراین با این شبکه (که قدری از شبکه‌ی mnist قوی‌تر بود) و با ۱۲ دور یادگیری و batch size برابر ۲۰۰ می‌توان به دقت ۸۸ درصد رسید.

جمع‌بندی

نکته‌ای که در اینجا وجود دارد این است که در بخش‌های رگرسیون، بهبود شبکه راحت بود. با افزایش تعداد نقاط و پیچیده کردن شبکه و قدری افزایش یادگیری، تقریبا همیشه امکان کاهش خطا وجود داشت. ولی کلاس‌بندی روی این دیتاست‌ها به این سادگی نیست و برای بهبود آن فراتر از یک حدی، نیازمند تکنیک‌های خاص و استفاده از شبکه‌های پیشرفته‌تر هستیم که فراتر از بحث ماست. واقعیتی که وجود دارد هم این است که این دیتاست‌ها ذاتا دارای نویز هستند، یعنی باید مقایسه‌ی ما با بخش رگرسیون نویزدار باشد، و نمی‌توان از یک دقتی فراتر رفت، یعنی همانطور که وقتی نویز گاوسی داریم MSE نمی‌تواند کمتر از یک حدی شود (بدون overfit) در اینجا هم دقت نمی‌تواند بیشتر از یک حدی شود زیرا دیتاست قطعا شامل تصاویری است که ابهام دارند (مثلا عدد ۹ی که خیلی شبیه ۰ باشد یا شلواری که خیلی شبیه دامن باشد) و هرگز نمی‌توان دقت شبکه را از حدی بالاتر برد.

بخش ششم: کاهش نویز

در این بخش از دیتاست MNIST استفاده می‌کنیم و دیتاست Fashion MNIST را به کنار می‌گذاریم. البته برای محاسبه‌ی عددی خطا، هردو مناسب هستند، ولی چون قصد داریم با چشم هم نتیجه را مشاهده کنیم، قضاوت این که یکی از تصاویر Fashion MNIST دقیقا چیست و چه زمانی نتیجه خوب است، سخت است.

توضیح کد

```python
def noisify(image, intensity):

    if intensity > 0:
        image_copy = random_noise(image, mode='gaussian', clip=True)
    if intensity > 1:
        image_copy = random_noise(image_copy, mode='gaussian', clip=True)
        image_copy = random_noise(image_copy, mode='salt', clip=True)
    if intensity > 2:
        image_copy = random_noise(image_copy, mode='gaussian', clip=True)
        image_copy = random_noise(image_copy, mode='salt', clip=True)
    if intensity > 3:
        image_copy = random_noise(image_copy, mode='gaussian', clip=True)
        image_copy = random_noise(image_copy, mode='salt', clip=True)
        image_copy = random_noise(image_copy, mode='gaussian', clip=True)
```

ابتدا یک تابع تعریف می‌کنیم که نویز را به یک تصویر اضافه کند. کتابخانه‌ی scikit-image شامل توابع آماده‌ای برای این کار است. متغیر intensity برای تعیین شدت نویز است. بر اساس مقدار آن، مرحله به مرحله نویز زیاد می‌شود. در مراحل دوم و بعدتر، هربار در کنار نویز gaussian، نویز salt هم داریم که برخی پیکسل‌ها را به صورت رندوم ۱ می‌کند.

```python
def add_image_to_plot(axs, row, img_n, img_dn, img_r):
    axs[row, 0].imshow(img_n)
    axs[row, 1].imshow(img_dn)
    axs[row, 2].imshow(img_r)
    return axs
```

این تابع سه تصویر معادل یک ردیف را به پلات ما اضافه می‌کند. تصویر اول حاوی نویز، تصویر دوم حاصل نویزگیری با شبکه و تصویر سوم، ground truth است که تصویر ما قبل از نویزگذاری است.

```
32  # load dataset from keras
33  (input_train, output_train), (input_test, output_test) = mnist.load_data()
34
35  # extrack 10k images and generate noised samples
36  images = input_train[0:12000]/255
37
38  noise_intensity = 1
39
40  images_noised = np.array(list(map(lambda x: noisify(x, noise_intensity), images))).reshape(-1,
```

```
42  input_train = images_noised[0:10000]
43  output_train = images[0:10000]
44  input_test = images_noised[10000:12000]
45  output_test = images[10000:12000]
```

در این چند خط، دیتاست بارگذاری شده و ۱۲۰۰۰ عضو اول آن انتخاب می‌شوند. از آنجایی که چیدمان دیتاست رندوم است نیازی نیست اندیس‌های رندوم تولید کنیم. ۱۰۰۰۰ عضو اول برای یادگیری هستند. سپس نمونه‌ی نویزدار عکس نیز تولید می‌شود.

```
47  # create and train model
48  model = Sequential()
49  model.add(Dense(500, input_dim = 28 * 28, activation= 'relu'))
50  model.add(Dense(1000, activation = 'relu'))
51  model.add(Dense(28 * 28, activation = 'relu'))
52
53  model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mean_squared_error'])
54
55  model.fit(input_train.reshape(-1, 28*28), output_train.reshape(-1, 28*28), batch_size=32, epoc
56
```

در این قسمت مدل ایجاد، کامپایل و train می‌شود. تنها نکته این است که این بار دیتاست را در کل reshape نکردیم و موقع پاس دادن آن به keras این عمل روی آن انجام شد. همچنین اگرچه کار ما ذاتا رگرسیون است، ولی مقادیر پیکسل‌ها تقسیم بر ۲۵۵ شده و در نتیجه ReLU به خوبی کار ما را انجام می‌دهد و نیازی به نورون linear نداریم.

```
_, model_MSE = model.evaluate(input_test.reshape(-1, 28*28), output_test.reshape(-1, 28*28))
print('MSE: ' +str(model_MSE))
```

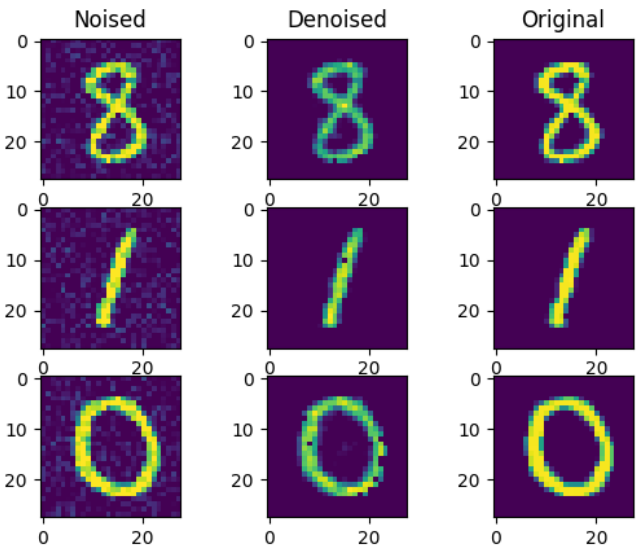
در این قسمت، مدل ارزیابی شده و MSE برای دادگان تست محاسبه می‌شود.

```
# plot results
fig = plt.figure(dpi=100)
axs = fig.subplots(3, 3)
i1, i2, i3 = randint(0, 12000), randint(0, 12000), randint(0, 12000)
add_image_to_plot(axs, 0, images_noised[i1],
                  model.predict(images_noised[i1].reshape(-1, 28*28)).reshape(28, 28),
                  images[i1])
add_image_to_plot(axs, 1, images_noised[i2],
                  model.predict(images_noised[i2].reshape(-1, 28*28)).reshape(28, 28),
                  images[i2])
add_image_to_plot(axs, 2, images_noised[i3],
                  model.predict(images_noised[i3].reshape(-1, 28*28)).reshape(28, 28),
                  images[i3])

axs[0, 0].set_title("Noised")
axs[0, 1].set_title("Denoised")
axs[0, 2].set_title("Original")

plt.show()
```

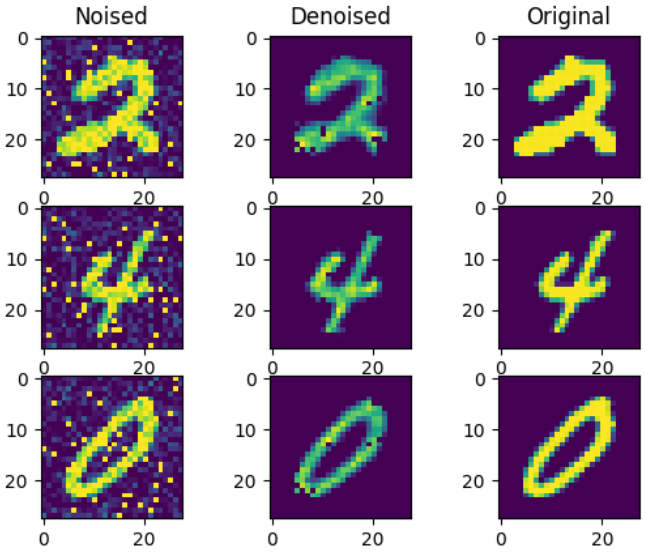
در نهایت سه اندیس تصادفی انتخاب شده و به ازای هر اندیس، تصویر اصلی، تصویر نویزدار شده و تصویری که نویز آن توسط شبکه حذف شده است مشاهده می‌شود.

بررسی چهار سطح نویز مختلف

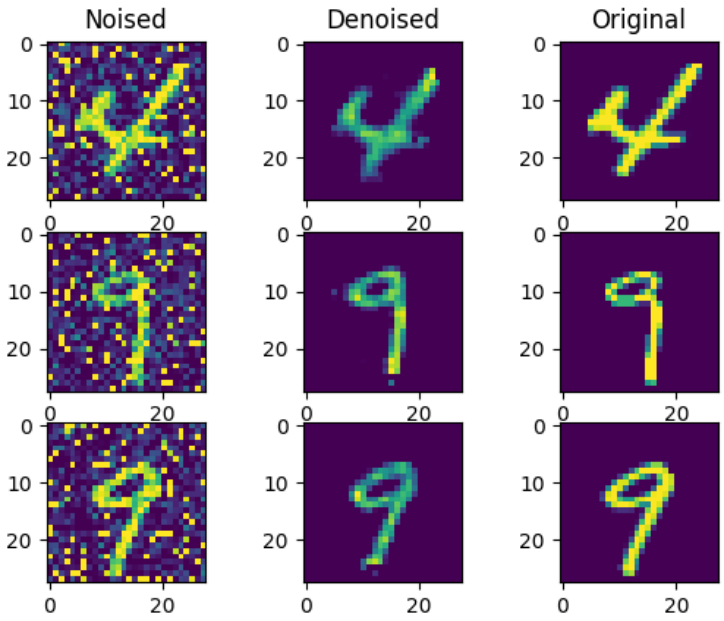
در این قسمت مشاهده می‌کنیم که با چهار سطح نویز مختلف و شبکه‌ای که در کد بالا مشاهده شد، چه نتیجه‌ای حاصل می‌شود.



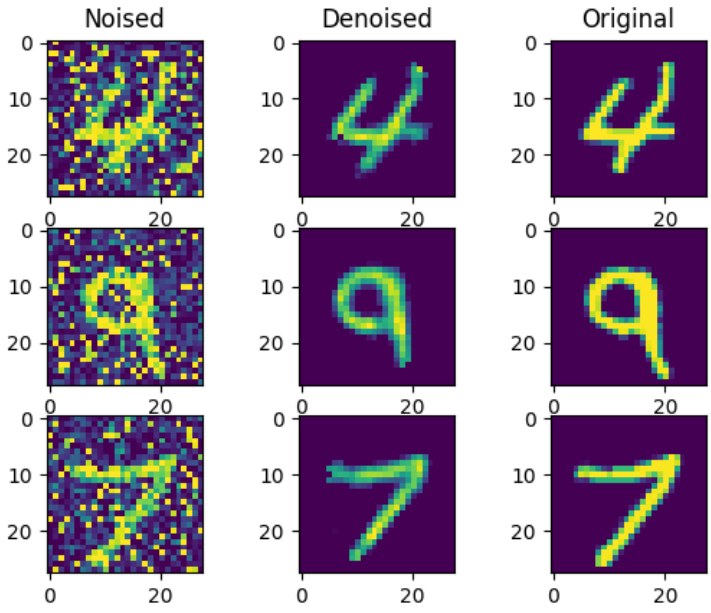
Noise Intensity=1, MSE=0.00832



Noise Intensity=2, MSE=0.01457



Noise Intensity=3, MSE=0.01512



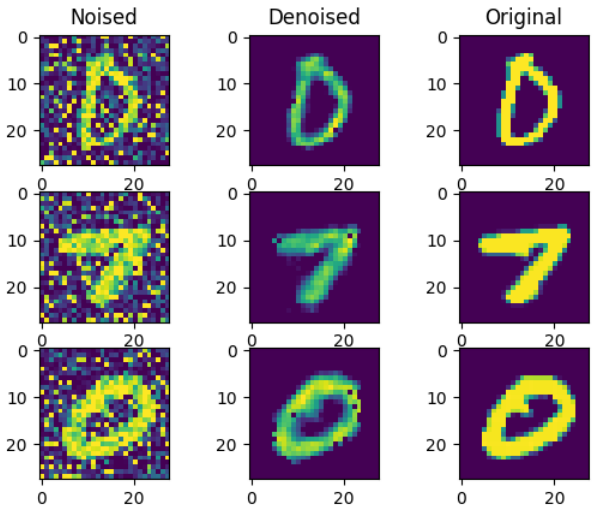
Noise Intensity=4, MSE=0.01788

بنابراین مشاهده می‌شود که شبکه‌ی عصبی توانایی کافی در رفع نویز را دارد. در ادامه تحلیل‌های بیشتری روی این موضوع انجام می‌دهیم. البته دقت داریم که این MSE ها کم نیستند — اعداد ورودی و خروجی در بازه‌ی ۰ تا ۱ هستند پس بدترین MSE ممکن برابر ۱ است!

مقایسه‌ی نتایج داده‌های آموزشی و آزمایشی

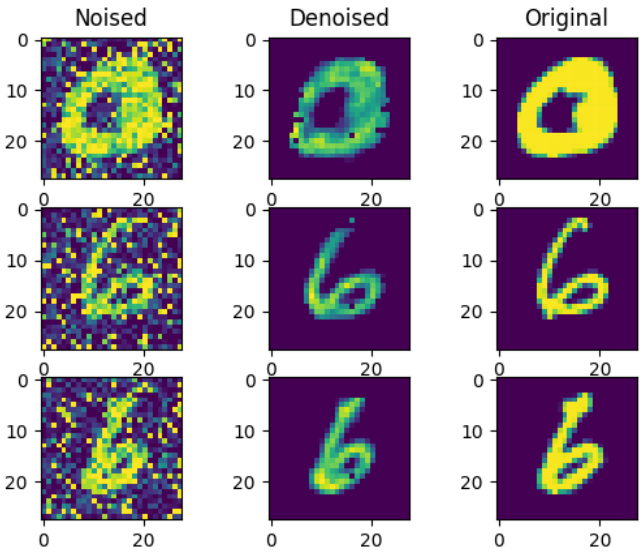
در قسمت قبل، نمونه‌های مشاهده شده بصورت مخلوط از نمونه‌های آموزشی و آزمایشی بودند. در این قسمت این دو را از هم جدا می‌کنیم و نتایج را بررسی می‌کنیم. در سراسر تحلیل‌هایمان، سطح نویز حداکثر است.

نویزگیری روی نمونه‌های آموزشی:



MSE=0.01859

نویزگیری روی نمونه‌های آزمایشی:

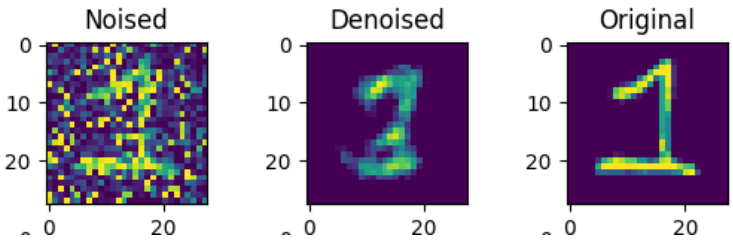


MSE=0.01860

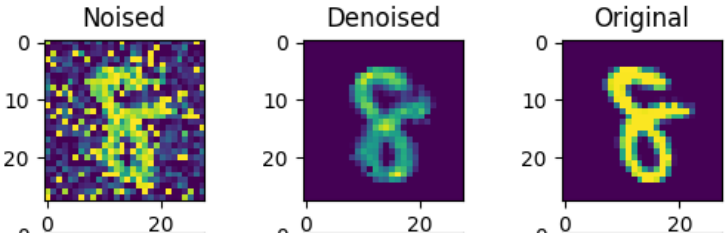
بنابراین شبکه به خوبی آموزش دیده است و تفاوت خاصی بین نمونه‌های آزمایشی و آموزشی نداریم.

تاثیر تعداد داده‌های آموزشی

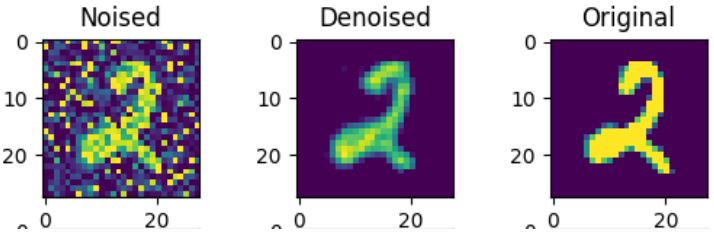
مشابه قسمت‌های قبلی، بررسی می‌کنیم که اندازه‌ی دیتاست چقدر در کارمان تاثیر دارد. به ترتیب برای سایز ۳۰۰۰ و ۵۰۰۰ و ۱۰۰۰۰ بررسی می‌کنیم که خطا چقدر است.



MSE=0.02214



MSE=0.02072

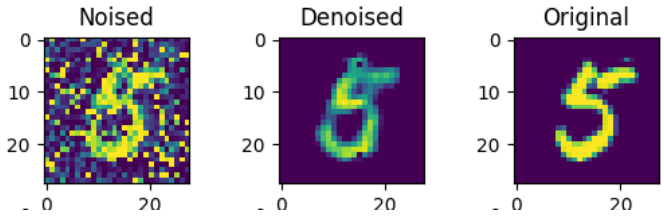


MSE=0.01823

مشاهده می‌شود که با ۳۰۰۰ داده وضعیت خیلی خوب نیست ولی بین ۵۰۰۰ و ۱۰۰۰۰ داده نتیجه خیلی فرق نمی‌کند. مجددا تاکید می‌شود که MSE=0.022 مقدار قابل توجهی است زیرا اعداد از ۰ تا ۱ هستند.

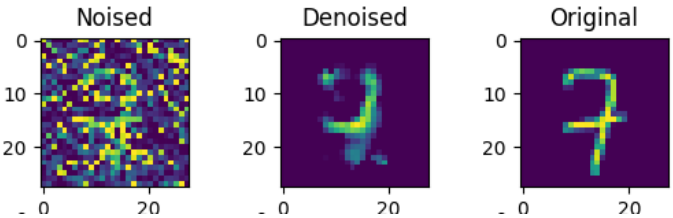
تعداد لایه‌های شبکه و تعداد نورون‌های هر لایه

۲۵۰ و ۵۰۰ و ۷۸۴ نورون



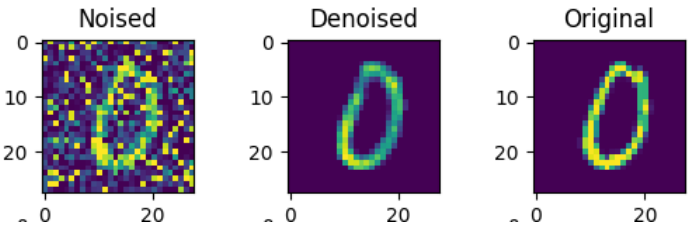
MSE=0.01830

۵۰۰ و ۱۰۰۰ و ۷۸۴ نورون



MSE=0.01840

۵۰۰ و ۱۰۰۰ و ۱۵۰۰ و ۷۸۴ نورون



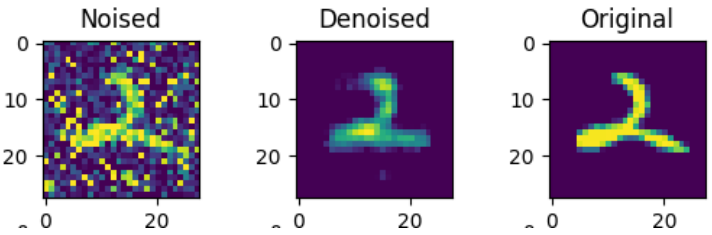
MSE = 0.01860

بنابراین به نظر می‌آید که مدل در حال overfit شدن است و نیازی نیست شبکه‌ی خیلی پیچیده‌ای داشته باشیم. شبکه‌ی اول این زیربخش که کمترین خطا را هم دارد، در حدود شبکه‌های بخش‌های ابتدایی پروژه است.

تعداد چرخه‌های شبکه برای تکمیل یادگیری

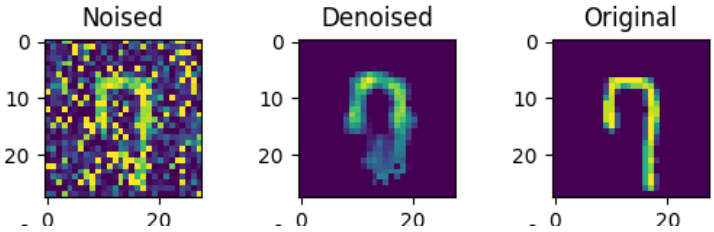
شبکه‌ای با ۳۰۰ و ۷۰۰ و ۷۸۴ نورون استفاده می‌کنیم و تعداد دورهای آموزشی را ۵ و ۱۰ و ۱۵ و ۲۰ و ۲۵ می‌گذاریم و نتیجه را مشاهده می‌کنیم:

با ۵ دور:



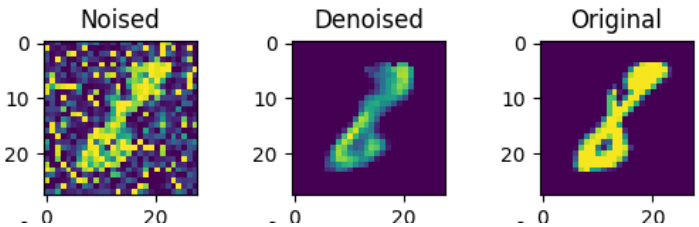
MSE = 0.01833

با ۱۰ دور:



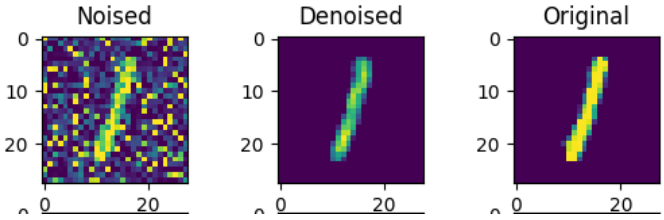
MSE = 0.01774

با ۱۵ دور:



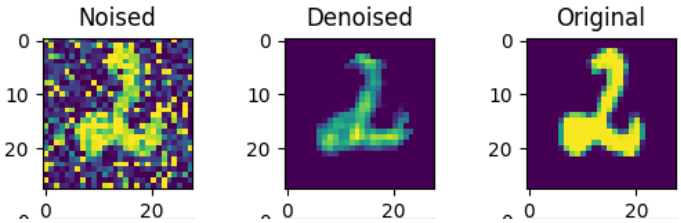
MSE=0.01827

با ۲۰ دور:



MSE=0.01777

با ۲۵ دور:

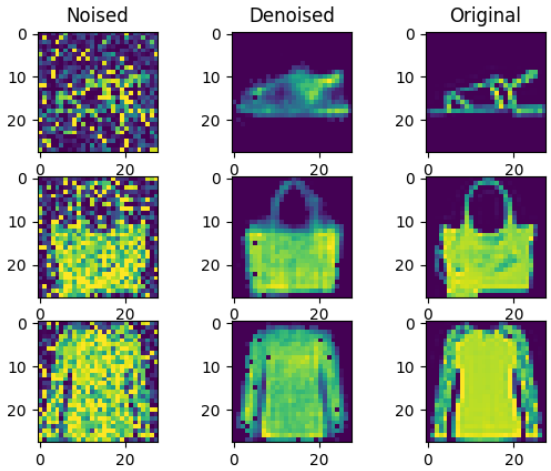


MSE=0.01774

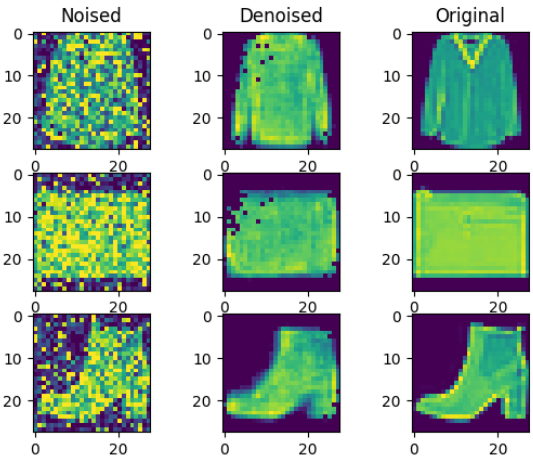
بنابراین اگرچه تعداد دور یادگیری زیادتر ذاتا چیز بهتری است، اما کمک زیادی به ما نمی‌کند، چون با چند برابر کردن مدت زمان یادگیری، خطایمان فقط به میزان کمی کاهش می‌یابد.

بررسی رفع نویز برای دیتاست Fashion MNIST

محض بررسی یک دیتاست پیچیده‌تر (که نویز گرفتن از آن سخت‌تر از نویز گرفتن از ارقام باشد) مجددا سراغ Fashion MNIST می‌رویم. با ۱۰ دور آموزشی و ۳۰۰ و ۷۰۰ نورون و ۱۰۰۰۰ داده‌ی آموزشی داریم:



MSE = 0.02013



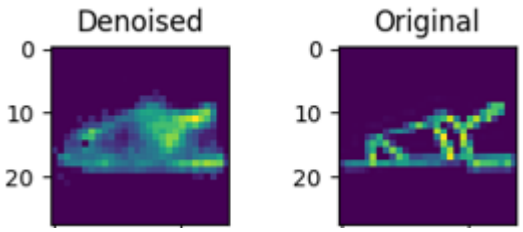
MSE = 0.01966

بنابراین می‌بینیم که کیفیت نسبت به دیتاست MNIST دچار افت شده است و تصاویر نویزگیری شده فرق قابل توجهی با تصاویر اصلی دارند.

علت این امر تا حدی بخاطر این است که ارقام دستنویس خیلی تنوع رنگ ندارند و پیکسل‌های آنها تقریبا صفر و یکی هستند، بنابراین اولا نویز خیلی آنها را خراب نمی‌کند و دوما رفع نویز ساده است. ولی در تصاویر لباس، جزئیات حائز اهمیت زیادتری داریم و رنگبندی زیاد است، پس هم نویز مخرب تر است و هم رفع آن سخت‌تر.

جمع‌بندی

در این قسمت رفع نویز با MLP بررسی شد. مشاهده شد که بدون سختی خاصی این کار برای دیتاست MNIST امکانپذیر است ولی برای Fashion MNIST سختتر است و برای دیتاست‌های واقعی‌تر حتی سختتر هم خواهد شد. دقت داریم که باید هدف از این رفع نویز هم مشخص باشد – چنانچه قصد داریم پس از رفع نویز inference انجام دهیم، کیفیت افت شدیدی نمی‌کند، ولی اگر بخواهیم از دید انسان عکس شبیه حالت اول شود، کار تا حدی سخت تر است. مثلا اگر به یک مثال از Fashion MNIST نگاه کنیم:



برای هردو تصویر واضح است که با یک کفش روبه‌رو هستیم و چنانچه این را ورودی شبکه عصبی قسمت پنج قرار دهیم قطعا جواب کفش خواهد بود، یعنی کلاس تصویر (از ۱۰ کلاس دیتاست) پس از نویز گرفتن و رفع نویز واضح است. این معادل این است که یک رقم مثلا ۳ بعد از نویز گرفتن و رفع نویز هنوز هم رقم ۳ باشد و به ۸ تبدیل نشده باشد.

ولی اگر از دید انسان به دو تصویر نگاه کنیم، تفاوت قابل توجه است. تصویر سمت چپ شبیه یک کتانی است و عملا دیگر تصویر سمت راست نیست و از دید یک انسان تصویر دچار تفاوت شدیدی شده است. علت این امر می‌تواند به وجود کتانی در داده‌های آموزشی شبکه نیز باشد.

جمع‌بندی

در شش قسمت این پروژه، رگرسیون و کلاس‌بندی با MLP بررسی شد. در هر قسمت سعی بر این بود که تاثیر تمامی مشخصات شبکه بررسی شود و مسیری که برای بهبود شبکه طی شده است توضیح داده شود. البته توضیحات کلیشه‌ای بیشتر در قسمت‌های اول آورده شد و در قسمت‌های بعدی از تکرار مکررات پرهیز شد و صرفا نتایج بررسی شدند. در چهار قسمت اول دیدیم که با شبکه‌ی عصبی چگونه می‌توان رگرسیون را روی توابع اجرا کرد و حتی توابع عجیب و چندضابطه‌ای را یاد گرفت. چالش اصلی این قسمت برون‌یابی شبکه روی توابع پیچیده بود – با بهبود شبکه می‌توان انتظار درون‌یابی خوبی داشت، ولی برون‌یابی توابع چندضابطه‌ای یا دوره‌ای خیلی سخت است. در قسمت پنجم کلاس‌بندی روی دیتاست MNIST و همچنین Fashion MNIST بررسی شد و دیدیم که یک شبکه به سادگی می‌تواند به دقت بالایی در کلاس‌بندی آنها دست پیدا کند. در قسمت ششم هم داده‌هایی از این دو دیتاست را آغشته به نویز کردیم و با شبکه‌ی عصبی نویز آنها را حذف کردیم.

منابع

1: https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

2: https://www.tensorflow.org/tutorials/keras/regression

3: https://github.com/titansarus/AI-Projects/tree/master/Neural%20Network

4: https://www.tensorflow.org/tutorials/keras/classification

5: https://keras.io/examples/vision/mnist_convnet/

6: https://www.analyticsvidhya.com/blog/2021/06/mnist-dataset-prediction-using-keras/

7: https://scikit-image.org/docs/stable/api/skimage.util.html#skimage.util.random_noise

8: پروژه‌ی ژنتیک خودم، که امکانات تولید دیتاست و نویز داشت